Python code for Artificial Intelligence

Foundations of Computational Agents

David L. Poole and Alan K. Mackworth

Version 0.9.17 of July 7, 2025.

https://aipython.org https://artint.info

Contents

Chapter 1

Python for Artificial Intelligence

AIPython contains runnable code for the book *Artificial Intelligence, foundations of computational agents, 3rd Edition* [Poole and Mackworth, 2023]. It has the following design goals:

- Readability is more important than efficiency, although the asymptotic complexity is not compromised. AIPython is not a replacement for well-designed libraries, or optimized tools. Think of it like a model of an engine made of glass, so you can see the inner workings; don't expect it to power a big truck, but it lets you see how an engine works to power a truck.

- It uses as few libraries as possible. A reader only needs to understand Python. Libraries hide details that we make explicit. The only library used is matplotlib for plotting and drawing.

1.1 Why Python?

We use Python because Python programs can be close to pseudo-code. It is designed for humans to read.

Python is reasonably efficient. Efficiency is usually not a problem for small examples. If your Python code is not efficient enough, a general procedure to improve it is to find out what is taking most of the time, and implement just that part more efficiently in some lower-level language. Many lower-level languages interoperate with Python nicely. This will result in much less programming and more efficient code (because you will have more time to optimize) than writing everything in a lower-level language. Much of the code here is more efficiently implemented in libraries that are more difficult to understand.

1.2 Getting Python

You need Python 3.9 or later (`https://python.org/`) and a compatible version of matplotlib (`https://matplotlib.org/`). This code is *not* compatible with Python 2 (e.g., with Python 2.7).

Download and install the latest Python 3 release from `https://python.org/` or `https://www.anaconda.com/download` (free download includes many libraries). This should also install `pip`. You can install matplotlib using

```
pip install matplotlib
```

in a terminal shell (not in Python). That should "just work". If not, try using `pip3` instead of `pip`.

The command `python` or `python3` should then start the interactive Python shell. You can quit Python with a control-D or with `quit()`.

To upgrade matplotlib to the latest version (which you should do if you install a new version of Python) do:

```
pip install --upgrade matplotlib
```

We recommend using the enhanced interactive python **ipython** (`https://ipython.org/`) [Pérez and Granger, 2007]. To install ipython after you have installed python do:

```
pip install ipython
```

1.3 Running Python

We assume that everything is done with an interactive Python shell. You can either do this with an IDE, such as IDLE that comes with standard Python distributions, or just running ipython or python (or perhaps ipython3 or python3) from a shell.

Here we describe the most simple version that uses no IDE. If you download the zip file, and cd to the "`aipython`" folder where the .py files are, you should be able to do the following, with user input in bold. The first `python` command is in the operating system shell; the `-i` is important to enter interactive mode.

```
python -i searchGeneric.py
Testing problem 1:
7 paths have been expanded and 4 paths remain in the frontier
Path found: A --> C --> B --> D --> G
Passed unit test
>>> searcher2 = AStarSearcher(searchProblem.acyclic_delivery_problem) #A*
>>> searcher2.search()  # find first path
16 paths have been expanded and 5 paths remain in the frontier
o103 --> o109 --> o119 --> o123 --> r123
>>> searcher2.search()  # find next path
```

```
21 paths have been expanded and 6 paths remain in the frontier
o103 --> b3 --> b4 --> o109 --> o119 --> o123 --> r123
>>> searcher2.search()  # find next path
28 paths have been expanded and 5 paths remain in the frontier
o103 --> b3 --> b1 --> b2 --> b4 --> o109 --> o119 --> o123 --> r123
>>> searcher2.search()  # find next path
No (more) solutions. Total of 33 paths expanded.
>>>
```

You can then interact at the last prompt.

There are many textbooks for Python. The best source of information about python is `https://www.python.org/`. The documentation is at `https://docs.python.org/3/`.

The rest of this chapter is about what is special about the code for AI tools. We only use the standard Python library and matplotlib. All of the exercises can be done (and should be done) without using other libraries; the aim is for you to spend your time thinking about how to solve the problem rather than searching for pre-existing solutions.

1.4 Pitfalls

It is important to know when side effects occur. Often AI programs consider what would/might happen given certain conditions. In many such cases, we don't want side effects. When an agent acts in the world, side effects are appropriate.

In Python, you need to be careful to understand side effects. For example, the inexpensive function to add an element to a list, namely `append`, changes the list. In a functional language like Haskell or Lisp, adding a new element to a list, without changing the original list, is a cheap operation. For example if x is a list containing n elements, adding an extra element to the list in Python (using `append`) is fast, but it has the side effect of changing the list x. To construct a new list that contains the elements of x plus a new element, without changing the value of x, entails copying the list, or using a different representation for lists. In the searching code, we will use a different representation for lists for this reason.

1.5 Features of Python

1.5.1 f-strings

Python can use matching `'`, `"`, `'''` or `"""`, the latter two respecting line breaks in the string. We use the convention that when the string denotes a unique symbol, we use single quotes, and when it is designed to be for printing, we use double quotes.

We make extensive use of f-strings https://docs.python.org/3/tutorial/inputoutput.html. In its simplest form

```
f"str1{e1}str2{e2}str3"
```

where e1 and e2 are expressions, is an abbreviation for

```
"str1"+str(e1)+"str2"+str(e2)+"str3"
```

where + is string concatenation, and str is a function that returns a string representation of its argument.

1.5.2 Lists, Tuples, Sets, Dictionaries and Comprehensions

We make extensive uses of lists, tuples, sets and dictionaries (dicts). See https://docs.python.org/3/library/stdtypes.html. Lists use "$[\dots]$", dictionaries use "$\{key : value, \dots\}$", sets use "$\{\dots\}$" (without the :), tuples use "$(\dots)$".

One of the nice features of Python is the use of **comprehensions**: list, tuple, set and dictionary comprehensions.

A list comprehension is of the form

$[fe$ `for` e `in` $iter$ `if` $cond]$

is the list values *fe* for each *e* in *iter* for which *cond* is true. The "`if` *cond*" part is optional, but the "`for`" and "`in`" are not optional. Here *e* is a variable (or a pattern that can be on the left side of =), *iter* is an iterator, which can generate a stream of data, such as a list, a set, a range object (to enumerate integers between ranges) or a file. *cond* is an expression that evaluates to either True or False for each *e*, and *fe* is an expression that will be evaluated for each value of *e* for which *cond* returns `True`. For example:

```
>>> [e*e for e in range(20) if e%2==0]
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

Comprehensions can also be used for sets and dictionaries. For example, the following creates an index for list a:

```
>>> a = ["a","f","bar","b","a","aaaaa"]
>>> ind = {a[i]:i for i in range(len(a))}
>>> ind
{'a': 4, 'f': 1, 'bar': 2, 'b': 3, 'aaaaa': 5}
>>> ind['b']
3
```

which means that 'b' is the element with index 3 in the list.

The assignment of ind could have also be written as:

```
>>> ind = {val:i for (i,val) in enumerate(a)}
```

where enumerate is a built-in function that, given a dictionary, returns an generator of $(index, value)$ pairs.

1.5.3 Generators

Python has generators which can be used for a form of lazy evaluation – only computing values when needed.

A comprehension in round parentheses gives a generator that can generate the elements as needed. The result can go in a list or used in another comprehension, or can be called directly using `next`. The procedure `next` takes an iterator and returns the next element (advancing the iterator); it raises a StopIteration exception if there is no next element. The following shows a simple example, where user input is prepended with >>>

```
>>> a = (e*e for e in range(20) if e%2==0)
>>> next(a)
0
>>> next(a)
4
>>> next(a)
16
>>> list(a)
[36, 64, 100, 144, 196, 256, 324]
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Notice how `list(a)` continued on the enumeration, and got to the end of it.

To make a procedure into a generator, the `yield` command returns a value that is obtained with `next`. It is typically used to enumerate the values for a `for` loop or in generators. (The `yield` command can also be used for coroutines, but AIPython only uses it for generators.)

A version of the built-in `range`, with 2 or 3 arguments (and positive steps) can be implemented as:[1]

pythonDemo.py — Some tricky examples

```
11 def myrange(start, stop, step=1):
12     """enumerates the values from start in steps of size step that are
13     less than stop.
14     """
15     assert step>0, f"only positive steps implemented in myrange: {step}"
16     i = start
17     while i<stop:
18         yield i
19         i += step
20
21 print("list(myrange(2,30,3)):",list(myrange(2,30,3)))
```

[1] Numbered lines are Python code available in the code-directory, `aipython`. The name of the file is given in the gray text above the listing. The numbers correspond to the line numbers in that file.

The built-in `range` is unconventional in how it handles a single argument, as the single argument acts as the second argument of the function. The built-in `range` also allows for indexing (e.g., `range(2,30,3)[2]` returns 8), but the above implementation does not. However `myrange` also works for floats, whereas the built-in `range` does not.

Exercise 1.1 Implement a version of `myrange` that acts like the built-in version when there is a single argument. (Hint: make the second argument have a default value that can be recognized in the function.) There is no need to make it work with indexing.

Yield can be used to generate the same sequence of values as in the example above.

pythonDemo.py — (continued)

```
23 def ga(n):
24     """generates square of even nonnegative integers less than n"""
25     for e in range(n):
26         if e%2==0:
27             yield e*e
28 a = ga(20)
```

The sequence of next(a), and list(a) gives exactly the same results as the comprehension at the start of this section.

It is straightforward to write a version of the built-in `enumerate` called `myenumerate`:

pythonDemo.py — (continued)

```
30 def myenumerate(iter, start=0):
31     i = start
32     for e in iter:
33         yield i,e
34         i += 1
```

1.5.4 Functions as first-class objects

Python can create lists and other data structures that contain functions. There is an issue that tricks many newcomers to Python. For a local variable in a function, the function uses the last value of the variable when the function is *called*, not the value of the variable when the function was defined (this is called "late binding"). This means if you want to use the value a variable has when the function is created, you need to save the current value of that variable. Whereas Python uses "late binding" by default, the alternative that newcomers often expect is "early binding", where a function uses the value a variable had when the function was defined. The following examples show how early binding can be implemented.

Consider the following programs designed to create a list of 5 functions, where the ith function in the list is meant to add i to its argument:

pythonDemo.py — (continued)

```
36 fun_list1 = []
37 for i in range(5):
38     def fun1(e):
39         return e+i
40     fun_list1.append(fun1)
41
42 fun_list2 = []
43 for i in range(5):
44     def fun2(e,iv=i):
45         return e+iv
46     fun_list2.append(fun2)
47
48 fun_list3 = [lambda e: e+i for i in range(5)]
49
50 fun_list4 = [lambda e,iv=i: e+iv for i in range(5)]
51
52 i=56
```

Try to predict, and then test to see the output, of the output of the following calls, remembering that the function uses the latest value of any variable that is not bound in the function call:

pythonDemo.py — (continued)

```
54 # in Shell do
55 ## ipython -i pythonDemo.py
56 # Try these (copy text after the comment symbol and paste in the Python
       prompt):
57 # print([f(10) for f in fun_list1])
58 # print([f(10) for f in fun_list2])
59 # print([f(10) for f in fun_list3])
60 # print([f(10) for f in fun_list4])
```

In the first for-loop, the function `fun1` uses `i`, whose value is the last value it was assigned. In the second loop, the function `fun2` uses `iv`. There is a separate `iv` variable for each function, and its value is the value of `i` when the function was defined. Thus `fun1` uses late binding, and `fun2` uses early binding. `fun_list3` and `fun_list4` are equivalent to the first two (except `fun_list4` uses a different `i` variable).

One of the advantages of using the embedded definitions (as in `fun1` and `fun2` above) over the lambda is that is it possible to add a `__doc__` string, which is the standard for documenting functions in Python, to the embedded definitions.

1.6 Useful Libraries

1.6.1 Timing Code

In order to compare algorithms, you may want to compute how long a program takes to run; this is called the **run time** of the program. The most straightforward way to compute the run time of `foo.bar(aaa)` is to use `time.perf_counter()`, as in:

```
import time
start_time = time.perf_counter()
foo.bar(aaa)
end_time = time.perf_counter()
print("Time:", end_time - start_time, "seconds")
```

Note that `time.perf_counter()` measures clock time; so this should be done without user interaction between the calls. On the interactive python shell, you should do:

```
start_time = time.perf_counter(); foo.bar(aaa); end_time = time.perf_counter()
```

If this time is very small (say less than 0.2 second), it is probably very inaccurate; run your code multiple times to get a more accurate count. For this you can use `timeit` (`https://docs.python.org/3/library/timeit.html`). To use timeit to time the call to `foo.bar(aaa)` use:

```
import timeit
time = timeit.timeit("foo.bar(aaa)",
                     setup="from __main__ import foo,aaa", number=100)
```

The setup is needed so that Python can find the meaning of the names in the string that is called. This returns the number of seconds to execute `foo.bar(aaa)` 100 times. The `number` should be set so that the run time is at least 0.2 seconds.

You should not trust a single measurement as that can be confounded by interference from other processes. `timeit.repeat` can be used for running `timeit` a few (say 3) times. When reporting the time of any computation, you should be explicit and explain what you are reporting. Usually the minimum time is the one to report (as it is the run with less interference).

1.6.2 Plotting: Matplotlib

The standard plotting for Python is matplotlib (`https://matplotlib.org/`). We will use the most basic plotting using the pyplot interface.

Here is a simple example that uses most of AIPython uses. The output is shown in Figure 1.1.

pythonDemo.py — (continued)

```
62 import matplotlib.pyplot as plt
63
```

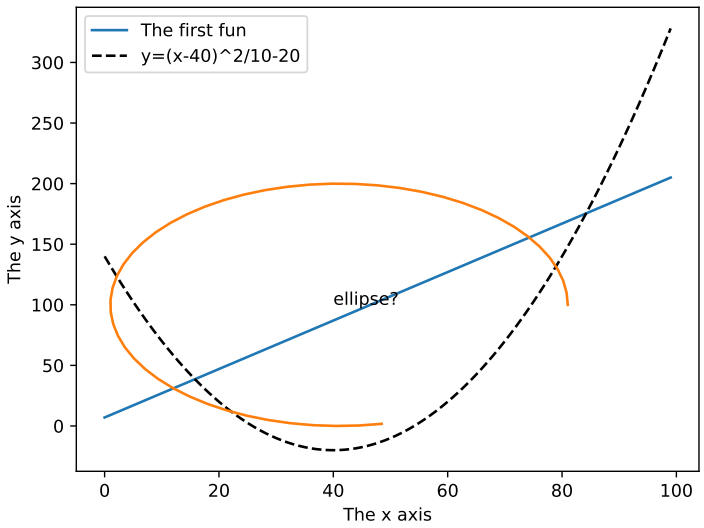


Figure 1.1: Result of pythonDemo code

```
64 def myplot(minv,maxv,step,fun1,fun2):
65     global fig, ax # allow them to be used outside myplot()
66     plt.ion() # make it interactive
67     fig, ax = plt.subplots()
68     ax.set_xlabel("The x axis")
69     ax.set_ylabel("The y axis")
70     ax.set_xscale('linear') # Makes a 'log' or 'linear' scale
71     xvalues = range(minv,maxv,step)
72     ax.plot(xvalues,[fun1(x) for x in xvalues],
73               label="The first fun")
74     ax.plot(xvalues,[fun2(x) for x in xvalues], linestyle='--',color='k',
75               label=fun2.__doc__) # use the doc string of the function
76     ax.legend(loc="upper right") # display the legend
77
78 def slin(x):
79     """y=2x+7"""
80     return 2*x+7
81 def sqfun(x):
82     """y=(x-40)^2/10-20"""
83     return (x-40)**2/10-20
84
85 # Try the following from shell:
86 # python -i pythonDemo.py
87 # myplot(0,100,1,slin,sqfun)
88 # ax.legend(loc="best")
89 # import math
90 # ax.plot([41+40*math.cos(th/10) for th in range(50)],
```

```
#         [100+100*math.sin(th/10) for th in range(50)])
# ax.text(40,100,"ellipse?")
# ax.set_xscale('log')
```

At the end of the code are some commented-out commands you should try in interactive mode. Cut from the file and paste into Python (and remember to remove the comments symbol and leading space).

1.7 Utilities

1.7.1 Display

To keep things simple, using only standard Python, AIPython code is written using a text-oriented tracing.

The method `self.display` is used to trace the program. Any call

`self.display`(*level*, *to_print* ...)

where the *level* is less than or equal to the value for `max_display_level` will be printed. The *to_print* ... can be anything that is accepted by the built-in `print` (including any keyword arguments).

The definition of `display` is:

display.py — A simple way to trace the intermediate steps of algorithms.

```
class Displayable(object):
    """Class that uses 'display'.
    The amount of detail is controlled by max_display_level
    """
    max_display_level = 1 # can be overridden in subclasses or instances

    def display(self,level,*args,**nargs):
        """print the arguments if level is less than or equal to the
        current max_display_level.
        level is an integer.
        the other arguments are whatever arguments print can take.
        """
        if level <= self.max_display_level:
            print(*args, **nargs) ##if error you are using Python2 not
                Python3
```

In this code, `args` gets a tuple of the positional arguments, and `nargs` gets a dictionary of the keyword arguments. This will not work in Python 2, and will give an error.

Any class that wants to use `display` can be made a subclass of `Displayable`.

To change the maximum display level to 3 for a class do:

Classname.`max_display_level` = 3

which will make calls to `display` in that class print when the value of `level` is less-than-or-equal to 3. The default display level is 1. It can also be changed for individual objects (the object value overrides the class value).

The value of `max_display_level` by convention is:

0 display nothing

1 display solutions (nothing that happens repeatedly)

2 also display the values as they change (little detail through a loop)

3 also display more details

4 and above even more detail

To implement a graphical user interface (GUI), the definition of `display` can be overridden. See, for example, `SearcherGUI` in Section 3.2.2 and `ConsistencyGUI` in Section 4.4.2. These GUIs use the AIPython code unchanged.

1.7.2 Argmax

Python has a built-in `max` function that takes a generator (or a list or set) and returns the maximum value. The `argmaxall` method takes a generator of (*element*, *value*) pairs, as for example is generated by the built-in `enumerate`(*list*) for lists or *dict*`.items()` for dictionaries. It returns a list of all elements with maximum value; `argmaxe` returns one of these values at random. The `argmax` method takes a list and returns the index of a random element that has the maximum value. `argmaxd` takes a dictionary and returns a key with maximum value.

utilities.py — AIPython useful utilities

```
11 import random
12 import math
13
14 def argmaxall(gen):
15     """gen is a generator of (element,value) pairs, where value is a real.
16     argmaxall returns a list of all of the elements with maximal value.
17     """
18     maxv = -math.inf     # negative infinity
19     maxvals = []    # list of maximal elements
20     for (e,v) in gen:
21         if v > maxv:
22             maxvals, maxv = [e], v
23         elif v == maxv:
24             maxvals.append(e)
25     return maxvals
26
27 def argmaxe(gen):
28     """gen is a generator of (element,value) pairs, where value is a real.
29     argmaxe returns an element with maximal value.
```

```
30      If there are multiple elements with the max value, one is returned at
            random.
31      """
32      return random.choice(argmaxall(gen))
33
34  def argmax(lst):
35      """returns maximum index in a list"""
36      return argmaxe(enumerate(lst))
37  # Try:
38  # argmax([1,6,3,77,3,55,23])
39
40  def argmaxd(dct):
41     """returns the arg max of a dictionary dct"""
42     return argmaxe(dct.items())
43  # Try:
44  # arxmaxd({2:5,5:9,7:7})
```

Exercise 1.2 Change `argmaxe` to have an optional argument that specifies whether you want the "first", "last" or a "random" index of the maximum value returned. If you want the first or the last, you don't need to keep a list of the maximum elements. Enable the other methods to have this optional argument, if appropriate.

1.7.3 Probability

For many of the simulations, we want to make a variable True with some probability. `flip(p)` returns True with probability `p`, and otherwise returns False.

utilities.py — (continued)

```
45  def flip(prob):
46      """return true with probability prob"""
47      return random.random() < prob
```

The `select_from_dist` method takes in a *item* : *probability* dictionary, and returns one of the items in proportion to its probability. The probabilities should sum to 1 or more. If they sum to more than one, the excess is ignored.

utilities.py — (continued)

```
49  def select_from_dist(item_prob_dist):
50      """ returns a value from a distribution.
51      item_prob_dist is an item:probability dictionary, where the
52          probabilities sum to 1.
53      returns an item chosen in proportion to its probability
54      """
55      ranreal = random.random()
56      for (it,prob) in item_prob_dist.items():
57          if ranreal < prob:
58              return it
59          else:
60              ranreal -= prob
61      raise RuntimeError(f"{item_prob_dist} is not a probability
            distribution")
```

1.8 Testing Code

It is important to test code early and test it often. We include a simple form of **unit test**. In your code, you should do more substantial testing than done here. Make sure you should also test boundary cases.

The following code tests `argmax`, but only if `utilities` is loaded in the top-level. If it is loaded in a module the test code is not run. The value of the current module is in `__name__` and if the module is run at the top-level, its value is `"__main__"`. See `https://docs.python.org/3/library/__main__.html`.

utilities.py — (continued)

```
63 def test():
64     """Test part of utilities"""
65     assert argmax([1,6,55,3,55,23]) in [2,4]
66     print("Passed unit test in utilities")
67     print("run test_aipython() to test (almost) everything")
68
69 if __name__ == "__main__":
70     test()
```

The following imports all of the python code and does a simple check of all of AIPython that has automatic checks. If you develop new algorithms or tests, add them here!

utilities.py — (continued)

```
72 def test_aipython():
73     import pythonDemo, display
74     # Agents: currently no tests
75     import agents, agentBuying, agentEnv, agentMiddle, agentTop,
           agentFollowTarget
76     # Search:
77     print("***** testing Search *****")
78     import searchGeneric, searchBranchAndBound, searchExample, searchTest
79     searchGeneric.test(searchGeneric.AStarSearcher)
80     searchBranchAndBound.test(searchBranchAndBound.DF_branch_and_bound)
81     searchTest.run(searchExample.problem1,"Problem 1")
82     import searchGUI, searchMPP, searchGrid
83     # CSP
84     print("\n***** testing CSP *****")
85     import cspExamples, cspDFS, cspSearch, cspConsistency, cspSLS
86     cspExamples.test_csp(cspDFS.dfs_solve1)
87     cspExamples.test_csp(cspSearch.solver_from_searcher)
88     cspExamples.test_csp(cspConsistency.ac_solver)
89     cspExamples.test_csp(cspConsistency.ac_search_solver)
90     cspExamples.test_csp(cspSLS.sls_solver)
91     cspExamples.test_csp(cspSLS.any_conflict_solver)
92     import cspConsistencyGUI, cspSoft
93     # Propositions
94     print("\n***** testing Propositional Logic *****")
```

```
    import logicBottomUp, logicTopDown, logicExplain, logicAssumables,
        logicNegation
    logicBottomUp.test()
    logicTopDown.test()
    logicExplain.test()
    logicNegation.test()
    # Planning
    print("\n***** testing Planning *****")
    import stripsHeuristic
    stripsHeuristic.test_forward_heuristic()
    stripsHeuristic.test_regression_heuristic()
    import stripsCSPPlanner, stripsPOP
    # Learning
    print("\n***** Learning with no inputs *****")
    import learnProblem, learnNoInputs, learnDT, learnLinear
    learnNoInputs.test_no_inputs(training_sizes=[4])
    data = learnProblem.Data_from_file('data/carbool.csv', one_hot=True,
        target_index=-1, seed=123)
    print("\n***** Decision Trees *****")
    learnDT. DT_learner(data).evaluate()
    print("\n***** Linear Learning *****")
    learnLinear.Linear_learner(data).evaluate()
    import learnCrossValidation, learnBoosting
    # Deep Learning
    import learnNN
    print("\n***** testing Neural Network Learning *****")
    learnNN.NN_from_arch(data, arch=[3]).evaluate()
    # Uncertainty
    print("\n***** testing Uncertainty *****")
    import probGraphicalModels, probRC, probVE, probStochSim
    probGraphicalModels.InferenceMethod.testIM(probRC.ProbSearch)
    probGraphicalModels.InferenceMethod.testIM(probRC.ProbRC)
    probGraphicalModels.InferenceMethod.testIM(probVE.VE)
    probGraphicalModels.InferenceMethod.testIM(probStochSim.RejectionSampling,
        threshold=0.1)
    probGraphicalModels.InferenceMethod.testIM(probStochSim.LikelihoodWeighting,
        threshold=0.1)
    probGraphicalModels.InferenceMethod.testIM(probStochSim.ParticleFiltering,
        threshold=0.1)
    probGraphicalModels.InferenceMethod.testIM(probStochSim.GibbsSampling,
        threshold=0.1)
    import probHMM, probLocalization, probDBN
    # Learning under uncertaint
    print("\n***** Learning under Uncertainty *****")
    import learnBayesian, learnKMeans, learnEM
    learnKMeans.testKM()
    learnEM.testEM()
    # Causality: currently no tests
    import probDo, probCounterfactual
    # Planning under uncertainty
```

```
    print("\n***** Planning under Uncertainty *****")
    import decnNetworks
    decnNetworks.test(decnNetworks.fire_dn)
    import mdpExamples
    mdpExamples.test_MDP(mdpExamples.partyMDP)
    import mdpGUI
    # Reinforcement Learning:
    print("\n***** testing Reinforcement Learning *****")
    import rlQLearner
    rlQLearner.test_RL(rlQLearner.Q_learner, alpha_fun=lambda k:10/(9+k))
    import rlQExperienceReplay
    rlQLearner.test_RL(rlQExperienceReplay.Q_ER_learner, alpha_fun=lambda
        k:10/(9+k))
    import rlStochasticPolicy
    rlQLearner.test_RL(rlStochasticPolicy.StochasticPIAgent,
        alpha_fun=lambda k:10/(9+k))
    import rlModelLearner
    rlQLearner.test_RL(rlModelLearner.Model_based_reinforcement_learner)
    import rlFeatures
    rlQLearner.test_RL(rlFeatures.SARSA_LFA_learner,
        es_kwargs={'epsilon':1}, eps=4)
    import rlQExperienceReplay, rlModelLearner, rlFeatures, rlGUI
    # Multiagent systems: currently no tests
    import rlStochasticPolicy, rlGameFeature
    # Individuals and Relations
    print("\n***** testing Datalog and Logic Programming *****")
    import relnExamples
    relnExamples.test_ask_all()
    # Knowledge Graphs and Ontologies
    print("\n***** testing Knowledge Graphs and Ontologies *****")
    import knowledgeGraph, knowledgeReasoning
    knowledgeGraph.test_kg()
    # Relational Learning: currently no tests
    import relnCollFilt, relnProbModels
    print("\n***** End of Testing*****")
```

Chapter 2

Agent Architectures and Hierarchical Control

This implements the controllers described in Chapter 2 of Poole and Mackworth [2023]. It defines an architecture that is also used by reinforcement learning (Chapter 13) and multiagent learning (Section 14.2).

AIPython only provides sequential implementations of the control. More sophisticated version may have them run concurrently. Higher-levels call lower-levels. The higher-levels calling the lower-level works in simulated environments where the lower-level are written to make sure they return (and don't go on forever), and the higher level doesn't take too long (as the lower-levels will wait until called again). More realistic architecture have the layers running concurrently so the lower layer can keep reacting while the higher layers are carrying out more complex computation.

2.1 Representing Agents and Environments

Both agents and the environment are treated as objects in the sense of object-oriented programming, with an internal state they maintain, and can evaluate methods. In this chapter, only a single agent is allowed; Section 14.2 allows for multiple agents.

An **environment** takes in actions of the agents, updates its internal state and returns the next percept, using the method `do`.

An **agent** implements the method `select_action` that takes a percept and returns the next action, updating its internal state as appropriate.

The methods `do` and `select_action` are chained together to build a simulator. Initially the simulator needs either an action or a percept. There are two variants used:

- An agent implements the `initial_action(percept)` method which is used initially. This is the method used in the reinforcement learning chapter (page 319).
- The environment implements the `initial_percept()` method which gives the initial percept for the agent. This is the method is used in this chapter.

The state of the agent and the state of the environment are represented using standard Python variables, which are updated as the state changes. The percept and the actions are represented as variable-value dictionaries.

`Agent` and `Environment` are subclasses of `Displayable` so that they can use the `display` method described in Section 1.7.1. `raise NotImplementedError()` is a way to specify an abstract method that needs to be overridden in any implemented agent or environment.

agents.py — Agent and Controllers

```
from display import Displayable

class Agent(Displayable):

    def initial_action(self, percept):
        """return the initial action."""
        return self.select_action(percept) # same as select_action

    def select_action(self, percept):
        """return the next action (and update internal state) given percept
        percept is variable:value dictionary
        """
        raise NotImplementedError("go") # abstract method
```

The environment implements a `do(action)` method where `action` is a variable-value dictionary. This returns a percept, which is also a variable-value dictionary. The use of dictionaries allows for structured actions and percepts.

Note that

agents.py — (continued)

```
class Environment(Displayable):
    def initial_percept(self):
        """returns the initial percept for the agent"""
        raise NotImplementedError("initial_percept") # abstract method

    def do(self, action):
        """does the action in the environment
        returns the next percept """
        raise NotImplementedError("Environment.do") # abstract method
```

The simulator is initialized with `initial_percept` and then the agent and the environment take turns in updating their states and returning the action and the percept. This simulator runs for n steps. A slightly more sophisticated simulator could run until some stopping condition.

agents.py — (continued)

```
class Simulate(Displayable):
    """simulate the interaction between the agent and the environment
    for n time steps.
    """
    def __init__(self, agent, environment):
        self.agent = agent
        self.env = environment
        self.percept = self.env.initial_percept()
        self.percept_history = [self.percept]
        self.action_history = []

    def go(self, n):
        for i in range(n):
            action = self.agent.select_action(self.percept)
            self.display(2,f"i={i} action={action}")
            self.percept = self.env.do(action)
            self.display(2,f"    percept={self.percept}")
```

2.2 Paper buying agent and environment

To run the demo, in folder "aipython", load "agents.py", using e.g., `ipython -i agentBuying.py`, and copy and paste the commented-out commands at the bottom of that file.

This is an implementation of Example 2.1 of Poole and Mackworth [2023]. You might get different plots to Figures 2.2 and 2.3 as there is randomness in the environment.

2.2.1 The Environment

The environment state is given in terms of the `time` and the amount of paper in `stock`. It also remembers the in-stock history and the price history. The percept consists of the price and the amount of paper in stock. The action of the agent is the number to buy.

Here we assume that the price changes are obtained from the `price_delta` list which gives the change in price for each time. When the time is longer than the list, it repeats the list. Note that the sum of the changes is greater than zero, so that prices tend to increase. There is also randomness (noise) added to the prices. The agent cannot access the price model; it just observes the prices and the amount in stock.

agentBuying.py — Paper-buying agent

```
import random
from agents import Agent, Environment, Simulate
from utilities import select_from_dist
```

```
14
15 class TP_env(Environment):
16     price_delta = [0, 0, 0, 21, 0, 20, 0, -64, 0, 0, 23, 0, 0, 0, -35,
17         0, 76, 0, -41, 0, 0, 0, 21, 0, 5, 0, 5, 0, 0, 0, 5, 0, -15, 0, 5,
18       0, 5, 0, -115, 0, 115, 0, 5, 0, -15, 0, 5, 0, 5, 0, 0, 0, 5, 0,
19       -59, 0, 44, 0, 5, 0, 5, 0, 0, 0, 5, 0, -65, 50, 0, 5, 0, 5, 0, 0,
20       0, 5, 0]
21     sd = 5 # noise standard deviation
22
23     def __init__(self):
24         """paper buying agent"""
25         self.time=0
26         self.stock=20
27         self.stock_history = [] # memory of the stock history
28         self.price_history = [] # memory of the price history
29
30     def initial_percept(self):
31         """return initial percept"""
32         self.stock_history.append(self.stock)
33         self.price = round(234+self.sd*random.gauss(0,1))
34         self.price_history.append(self.price)
35         return {'price': self.price,
36                 'instock': self.stock}
37
38     def do(self, action):
39         """does action (buy) and returns percept consisting of price and
               instock"""
40         used = select_from_dist({6:0.1, 5:0.1, 4:0.1, 3:0.3, 2:0.2, 1:0.2})
41         # used = select_from_dist({7:0.1, 6:0.2, 5:0.2, 4:0.3, 3:0.1,
               2:0.1}) # uses more paper
42         bought = action['buy']
43         self.stock = self.stock+bought-used
44         self.stock_history.append(self.stock)
45         self.time += 1
46         self.price = round(self.price
47                            + self.price_delta[self.time%len(self.price_delta)] #
                               repeating pattern
48                            + self.sd*random.gauss(0,1)) # plus randomness
49         self.price_history.append(self.price)
50         return {'price': self.price,
51                 'instock': self.stock}
```

2.2.2 The Agent

The agent does not have access to the price model but can only observe the current price and the amount in stock. It has to decide how much to buy.

The belief state of the agent is an estimate of the average price of the paper, and the total amount of money the agent has spent.

agentBuying.py — (continued)

```
class TP_agent(Agent):
    def __init__(self):
        self.spent = 0
        percept = env.initial_percept()
        self.ave = self.last_price = percept['price']
        self.instock = percept['instock']
        self.buy_history = []

    def select_action(self, percept):
        """return next action to carry out
        """
        self.last_price = percept['price']
        self.ave = self.ave+(self.last_price-self.ave)*0.05
        self.instock = percept['instock']
        if self.last_price < 0.9*self.ave and self.instock < 60:
            tobuy = 48
        elif self.instock < 12:
            tobuy = 12
        else:
            tobuy = 0
        self.spent += tobuy*self.last_price
        self.buy_history.append(tobuy)
        return {'buy': tobuy}
```

Set up an environment and an agent. Uncomment the last lines to run the agent for 90 steps, and determine the average amount spent.

agentBuying.py — (continued)

```
env = TP_env()
ag = TP_agent()
sim = Simulate(ag,env)
#sim.go(90)
#ag.spent/env.time ## average spent per time period
```

2.2.3 Plotting

The following plots the price and number in stock history:

agentBuying.py — (continued)

```
import matplotlib.pyplot as plt

class Plot_history(object):
    """Set up the plot for history of price and number in stock"""
    def __init__(self, ag, env):
        self.ag = ag
        self.env = env
        plt.ion()
        fig, self.ax = plt.subplots()
        self.ax.set_xlabel("Time")
        self.ax.set_ylabel("Value")
```

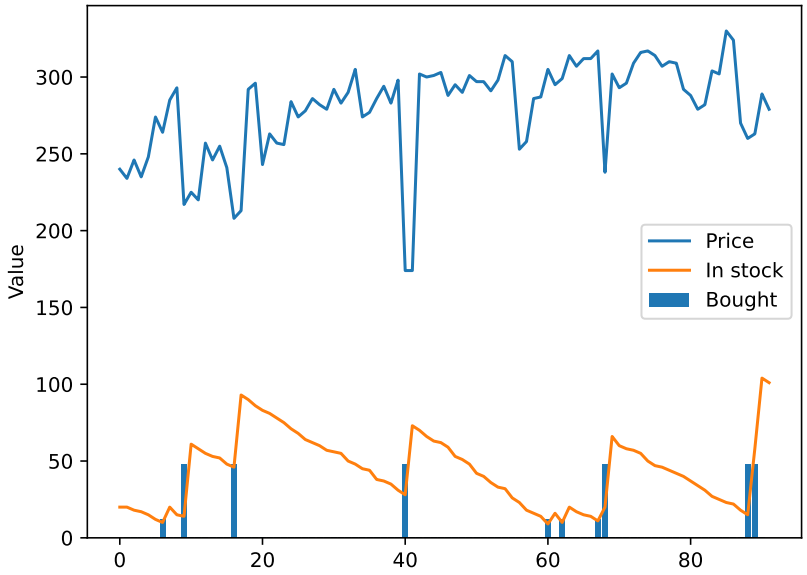


Figure 2.1: Percept and command traces for the paper-buying agent

```
94
95      def plot_env_hist(self):
96          """plot history of price and instock"""
97          num = len(env.stock_history)
98          self.ax.plot(range(num),env.price_history,label="Price")
99          self.ax.plot(range(num),env.stock_history,label="In stock")
100         self.ax.legend()
101
102     def plot_agent_hist(self):
103         """plot history of buying"""
104         num = len(ag.buy_history)
105         self.ax.bar(range(1,num+1), ag.buy_history, label="Bought")
106         self.ax.legend()
107
108 # sim.go(100); print(f"agent spent ${ag.spent/100}")
109 # pl = Plot_history(ag,env); pl.plot_env_hist(); pl.plot_agent_hist()
```

Figure 2.1 shows the result of the plotting in the previous code.

Exercise 2.1 Design a better controller for a paper-buying agent.

- Justify a performance measure that is a fair comparison. Note that minimizing the total amount of money spent may be unfair to agents who have built up a stockpile, and favors agents that end up with no paper.
- Give a controller that can work for many different price histories. An agent can use other local state variables, but does not have access to the environment model.

- Is it worthwhile trying to infer the amount of paper that the home uses? (Try your controller with the different paper consumption commented out in `TP_env.do`.)

2.3 Hierarchical Controller

To run the hierarchical controller, in folder "`aipython`", load "`agentTop.py`", using e.g., `ipython -i agentTop.py`, and copy and paste the commands near the bottom of that file.

In this implementation, each layer, including the top layer, implements the environment class, because each layer is seen as an environment from the layer above.

The robot controller is decomposed as follows. The world defines the walls. The body describes the robot's position, and its physical abilities such as whether its whisker sensor of on. The body can be told to steer left or right or to go straight. The middle layer can be told to go to *x*-*y* positions, avoiding walls. The top layer knows about named locations, such as the storage room and location `o103`, and their *x*-*y* positions. It can be told a sequence of locations, and tells the middle layer to go to the positions of the locations in turn.

2.3.1 Body

`Rob_body` defines everything about the agent body, its position and orientation and whether its whisker sensor is on. It implements the Environment class as it is treated as an environment by the higher layers. It can be told to turn left or right or to go straight.

agentEnv.py — Agent environment

```
11 import math
12 from agents import Environment
13 import matplotlib.pyplot as plt
14 import time
15
16 class Rob_body(Environment):
17     def __init__(self, world, init_pos=(0,0), init_dir=90):
18         """ world is the current world
19         init_pos is a pair of (x-position, y-position)
20         init_dir is a direction in degrees; 0 is to right, 90 is
               straight-up, etc
21         """
22         self.world = world
23         self.rob_pos = init_pos
24         self.rob_dir = init_dir
25         self.turning_angle = 18 # degrees that a left makes
26         self.whisker_length = 6 # length of the whisker
27         self.whisker_angle = 30 # angle of whisker relative to robot
```

```
28          self.crashed = False
29
30      def percept(self):
31          return {'rob_pos':self.rob_pos,
32                  'rob_dir':self.rob_dir, 'whisker':self.whisker(),
                        'crashed':self.crashed}
33      initial_percept = percept # use percept function for initial percept too
34
35      def do(self, action):
36          """ action is {'steer':direction}
37          direction is 'left', 'right' or 'straight'.
38          Returns current percept.
39          """
40          if self.crashed:
41              return self.percept()
42          direction = action['steer']
43          compass_deriv =
                {'left':1,'straight':0,'right':-1}[direction]*self.turning_angle
44          self.rob_dir = (self.rob_dir + compass_deriv +360)%360 # make in
                range [0,360)
45          x,y = self.rob_pos
46          rob_pos_new = (x + math.cos(self.rob_dir*math.pi/180),
47                         y + math.sin(self.rob_dir*math.pi/180))
48          path = (self.rob_pos,rob_pos_new)
49          if any(line_segments_intersect(path,wall) for wall in
                self.world.walls):
50              self.crashed = True
51          self.rob_pos = rob_pos_new
52          self.world.do({'rob_pos':self.rob_pos,
53                         'crashed':self.crashed, 'whisker':self.whisker()})
54          return self.percept()
```

The Boolean `whisker` method returns True when the the robots whisker sensor intersects with a wall.

agentEnv.py — (continued)

```
56      def whisker(self):
57          """returns true whenever the whisker sensor intersects with a wall
58          """
59          whisk_ang_world = (self.rob_dir-self.whisker_angle)*math.pi/180
60              # angle in radians in world coordinates
61          (x,y) = self.rob_pos
62          wend = (x + self.whisker_length * math.cos(whisk_ang_world),
63                  y + self.whisker_length * math.sin(whisk_ang_world))
64          whisker_line = (self.rob_pos, wend)
65          hit = any(line_segments_intersect(whisker_line,wall)
66                    for wall in self.world.walls)
67          return hit
68
69  def line_segments_intersect(linea, lineb):
70      """returns true if the line segments, linea and lineb intersect.
```

```
    A line segment is represented as a pair of points.
    A point is represented as a (x,y) pair.
    """
    ((x0a,y0a),(x1a,y1a)) = linea
    ((x0b,y0b),(x1b,y1b)) = lineb
    da, db = x1a-x0a, x1b-x0b
    ea, eb = y1a-y0a, y1b-y0b
    denom = db*ea-eb*da
    if denom==0:  # line segments are parallel
        return False
    cb = (da*(y0b-y0a)-ea*(x0b-x0a))/denom # intersect along line b
    if cb<0 or cb>1:
        return False  # intersect is outside line segment b
    ca = (db*(y0b-y0a)-eb*(x0b-x0a))/denom # intersect along line a
    return 0<=ca<=1 # intersect is inside both line segments

# Test cases:
# assert line_segments_intersect(((0,0),(1,1)),((1,0),(0,1)))
# assert not line_segments_intersect(((0,0),(1,1)),((1,0),(0.6,0.4)))
# assert line_segments_intersect(((0,0),(1,1)),((1,0),(0.4,0.6)))
```

2.3.2 Middle Layer

The middle layer acts like both a controller (for the body layer) and an environment for the upper layer. It has to tell the body how to steer. Thus it calls *env.do*(·), where env is the body. It implements do(\cdot) for the top layer, where the action specifies an *x*-*y* position to go to and a timeout.

agentMiddle.py — Middle Layer

```
from agents import Environment
import math

class Rob_middle_layer(Environment):
    def __init__(self, lower):
        """The lower-level for the middle layer is the body.
        """
        self.lower = lower
        self.percept = lower.initial_percept()
        self.straight_angle = 11 # angle that is close enough to straight
            ahead
        self.close_threshold = 1 # distance that is close enough to arrived
        self.close_threshold_squared = self.close_threshold**2 # just
            compute it once

    def initial_percept(self):
        return {}

    def do(self, action):
        """action is {'go_to':target_pos,'timeout':timeout}
```

```
29          target_pos is (x,y) pair
30          timeout is the number of steps to try
31          returns {'arrived':True} when arrived is true
32              or {'arrived':False} if it reached the timeout
33          """
34          if 'timeout' in action:
35              remaining = action['timeout']
36          else:
37              remaining = -1  # will never reach 0
38          target_pos = action['go_to']
39          arrived = self.close_enough(target_pos)
40          while not arrived and remaining != 0:
41              self.percept = self.lower.do({"steer":self.steer(target_pos)})
42              remaining -= 1
43              arrived = self.close_enough(target_pos)
44          return {'arrived':arrived}
```

The following method determines how to steer depending on whether the goal is to the right or the left of where the robot is facing.

agentMiddle.py — (continued)

```
46      def steer(self, target_pos):
47          if self.percept['whisker']:
48              self.display(3,'whisker on', self.percept)
49              return "left"
50          else:
51              return self.head_towards(target_pos)
52
53      def head_towards(self, target_pos):
54              """ given a target position, return the action that heads
                    towards that position
55              """
56              gx,gy = target_pos
57              rx,ry = self.percept['rob_pos']
58              goal_dir = math.acos((gx-rx)/math.sqrt((gx-rx)*(gx-rx)
59                                                     +(gy-ry)*(gy-ry)))*180/math.pi
60              if ry>gy:
61                  goal_dir = -goal_dir
62              goal_from_rob = (goal_dir - self.percept['rob_dir']+540)%360-180
63              assert -180 < goal_from_rob <= 180
64              if goal_from_rob > self.straight_angle:
65                  return "left"
66              elif goal_from_rob < -self.straight_angle:
67                  return "right"
68              else:
69                  return "straight"
70
71      def close_enough(self, target_pos):
72          """True when the robot's position is within close_threshold of
                target_pos
73          """
```

```
74          gx,gy = target_pos
75          rx,ry = self.percept['rob_pos']
76          return (gx-rx)**2 + (gy-ry)**2 <= self.close_threshold_squared
```

2.3.3 Top Layer

The top layer treats the middle layer as its environment. Note that the top layer is an environment for us to tell it what to visit.

agentTop.py — Top Layer

```
11  from display import Displayable
12  from agentMiddle import Rob_middle_layer
13  from agents import Agent, Environment
14
15  class Rob_top_layer(Agent, Environment):
16      def __init__(self, lower, world, timeout=200 ):
17          """lower is the lower layer
18          world is the world (which knows where the locations are)
19          timeout is the number of steps the middle layer goes before giving
                up
20          """
21          self.lower = lower
22          self.world = world
23          self.timeout = timeout # number of steps before the middle layer
                should give up
24
25      def do(self,plan):
26          """carry out actions.
27          actions is of the form {'visit':list_of_locations}
28          It visits the locations in turn.
29          """
30          to_do = plan['visit']
31          for loc in to_do:
32              position = self.world.locations[loc]
33              arrived = self.lower.do({'go_to':position,
                    'timeout':self.timeout})
34              self.display(1,"Goal",loc,arrived)
```

2.3.4 World

The world defines the walls and implements tracing.

agentEnv.py — (continued)

```
92  import math
93  from display import Displayable
94  import matplotlib.pyplot as plt
95
96  class World(Environment):
```

```
97      def __init__(self, walls = {}, locations = {},
            plot_size=(-10,120,-10,60)):
98          """walls is a set of line segments
99                where each line segment is of the form ((x0,y0),(x1,y1))
100          locations is a loc:pos dictionary
101             where loc is a named location, and pos is an (x,y) position.
102         """
103         self.walls = walls
104         self.locations = locations
105         self.loc2text = {}
106         self.history = [] # list of (pos, whisker, crashed)
107         # The following control how it is plotted
108         plt.ion()
109         fig, self.ax = plt.subplots()
110         #self.ax.set_aspect('equal')
111         self.ax.axis(plot_size)
112         self.sleep_time = 0.05  # time between actions (for real-time
                plotting)
113         self.draw()
114
115     def do(self, action):
116         """action is {'rob_pos':(x,y), 'whisker':Boolean, 'crashed':Boolean}
117         """
118         self.history.append((action['rob_pos'],action['whisker'],action['crashed']))
119         x,y = action['rob_pos']
120         if action['crashed']:
121             self.display(1, "*Crashed*")
122             self.ax.plot([x],[y],"r*",markersize=20.0)
123         elif action['whisker']:
124             self.ax.plot([x],[y],"ro")
125         else:
126             self.ax.plot([x],[y],"go")
127         plt.draw()
128         plt.pause(self.sleep_time)
129         return {'walls':self.walls}
```

2.3.5 Plotting

The following is used to plot the locations, the walls and (eventually) the movement of the robot. It can either plot the movement if the robot as it is going (with the default *env.plotting* = *True*), or not plot it as it is going (setting *env.plotting* = *False*; in this case the trace can be plotted using *pl.plot_run*()).

agentEnv.py — (continued)

```
131     def draw(self):
132         for wall in self.walls:
133             ((x0,y0),(x1,y1)) = wall
134             self.ax.plot([x0,x1],[y0,y1],"-k",linewidth=3)
135         for loc in self.locations:
```

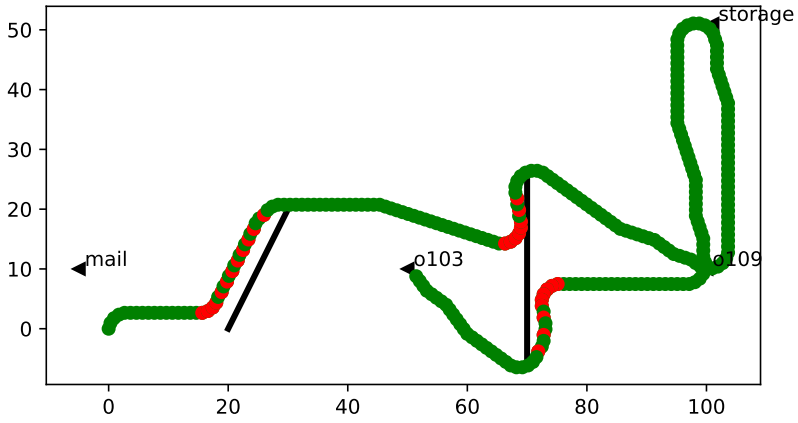


Figure 2.2: A trace of the trajectory of the agent. Red dots correspond to the whisker sensor being on; the green dot to the whisker sensor being off. The agent starts at position $(0,0)$ facing up.

```
136             self.plot_loc(loc)
137
138     def plot_loc(self, loc):
139         (x,y) = self.locations[loc]
140         if loc in self.loc2text:
141             for e in self.loc2text[loc]:
142                 e.remove() # e.set_visible(False)
143         self.loc2text[loc] = (
                self.ax.text(x,y,"*",ha="center",va="center",size=20),
144                         self.ax.text(x+2.0,y+1,loc)) # label above and to
                             the right
```

The following example shows a plot of the agent as it acts in the world. Figure 2.2 shows the result of the commented-out `top.do`

agentTop.py — (continued)

```
36 from agentEnv import Rob_body, World
37
38 def rob_ex():
39     global world, body, middle, top
40     world = World(walls = {((20,0),(30,20)), ((70,-5),(70,25))},
41                   locations = {'mail':(-5,10),
42                                'o103':(50,10),
                                     'o109':(100,10),'storage':(101,51)})
43     body = Rob_body(world)
44     middle = Rob_middle_layer(body)
45     top = Rob_top_layer(middle, world)
46
47 # try:
48 # top.do({'visit':['o109','storage','o109','o103']})
49 # You can directly control the middle layer:
```

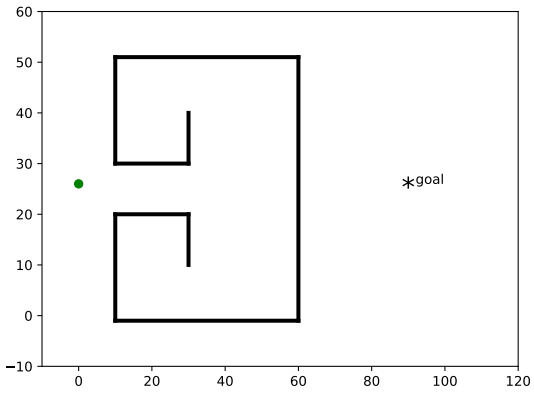


Figure 2.3: Robot trap

```
# middle.do({'go_to':(30,-5), 'timeout':200})
# Can you make it go around in circles?
# Can you make it crash?

if __name__ == "__main__":
    rob_ex()
    print("Try: top.do({'visit':['o109','storage','o109','o103']})")
```

Exercise 2.2 When does the robot go in circles? How could this be recognized and/or avoided?

Exercise 2.3 When does the agent crash? What sensor would avoid that? (Think about the worse configuration of walls.) Design a whisker-like sensor that never crashes (assuming it starts far enough from a wall) and allows the robot to go as close as possible to a wall.

Exercise 2.4 The following implements a robot trap (Figure 2.3). It is called a trap because, once it has hit the wall, it needs to follow the wall, but local features are not enough for it to know when it can head to the goal. Write a controller that can escape the "trap" and get to the goal. Would a beter sensor work? See Exercise 2.4 in the textbook for hints.

agentTop.py — (continued)

```
# Robot Trap for which the current controller cannot escape:
def robot_trap():
    global trap_world, trap_body, trap_middle, trap_top
    trap_world = World({((10, 51), (60, 51)), ((30, 10), (30, 20)),
                        ((10, -1), (10, 20)), ((10, 30), (10, 51)),
                        ((30, 30), (30, 40)), ((10, -1), (60, -1)),
                        ((10, 30), (30, 30)), ((10, 20), (30, 20)),
                        ((60, -1), (60, 51))},
                   locations={'goal':(90,25)})
```

```
    trap_body = Rob_body(trap_world,init_pos=(0,25), init_dir=90)
    trap_middle = Rob_middle_layer(trap_body)
    trap_top = Rob_top_layer(trap_middle, trap_world)

# Robot trap exercise:
# robot_trap()
# trap_body.do({'steer':'straight'})
# trap_top.do({'visit':['goal']})
# What if the goal was further to the right?
```

Plotting for Moving Targets

Exercise 2.5 of Poole and Mackworth [2023] refers to targets that can move. The following implements targets than can be moved using the mouse. To move a target using the mouse, press on the target, move it, and release at the desired location. This can be done while the animation is running.

agentFollowTarget.py — Plotting for moving targets

```
import matplotlib.pyplot as plt
from agentEnv import Rob_body, World
from agentMiddle import Rob_middle_layer
from agentTop import Rob_top_layer

class World_follow(World):
    def __init__(self, walls = {}, locations = {}, epsilon=5):
        """plot the agent in the environment.
        epsilon is the threshold how how close someone needs to click to
            select a location.
        """
        self.epsilon = epsilon
        World.__init__(self, walls, locations)
        self.canvas = self.ax.figure.canvas
        self.canvas.mpl_connect('button_press_event', self.on_press)
        self.canvas.mpl_connect('button_release_event', self.on_release)
        self.canvas.mpl_connect('motion_notify_event', self.on_move)
        self.pressloc = None
        for loc in self.locations:
            self.display(2,f"  loc {loc} at {self.locations[loc]}")

    def on_press(self, event):
        print("press", event)
        self.display(2,'v',end="")
        self.display(2,f"Press at ({event.xdata},{event.ydata}")
        self.pressloc = None
        if event.xdata:
          for loc in self.locations:
            lx,ly = self.locations[loc]
            if abs(event.xdata- lx) <= self.epsilon and abs(event.ydata-
                ly) <= self.epsilon :
```

```
40              self.display(2,f"moving {loc} from ({event.xdata},
                    {event.ydata})" )
41              self.pressloc = loc
42
43      def on_release(self, event):
44          self.display(2,'^',end="")
45          if self.pressloc is not None and event.xdata:
46              self.display(2,f"Placing {self.pressloc} at {(event.xdata,
                    event.ydata)}")
47              self.locations[self.pressloc] = (event.xdata, event.ydata)
48              self.plot_loc(self.pressloc)
49          self.pressloc = None
50
51      def on_move(self, event):
52          if self.pressloc is not None and event.inaxes:
53              self.display(2,'-',end="")
54              self.locations[self.pressloc] = (event.xdata, event.ydata)
55              self.plot_loc(self.pressloc)
56          else:
57              self.display(2,'.',end="")
58
59  def rob_follow():
60      global world, body, middle, top
61      world = World_follow(walls = {((20,0),(30,20)), ((70,-5),(70,25))},
62                   locations = {'mail':(-5,10), 'o103':(50,10),
63                                'o109':(100,10),'storage':(101,51)})
64      body = Rob_body(world)
65      middle = Rob_middle_layer(body)
66      top = Rob_top_layer(middle, world)
67
68  # top.do({'visit':['o109','storage','o109','o103']})
69
70  if __name__ == "__main__":
71      rob_follow()
72      print("Try: top.do({'visit':['o109','storage','o109','o103']})")
```

Exercise 2.5 Do Exercise 2.5 of Poole and Mackworth [2023].

Exercise 2.6 Change the code to also allow walls to move.

Chapter 3

Searching for Solutions

3.1 Representing Search Problems

A search problem consists of:

- a start node
- a *neighbors* function that given a node, returns an enumeration of the arcs from the node
- a specification of a goal in terms of a Boolean function that takes a node and returns true if the node is a goal
- a (optional) heuristic function that, given a node, returns a non-negative real number. The heuristic function defaults to zero.

As far as the searcher is concerned a node can be anything. If multiple-path pruning is used, a node must be `hashable`. In the simple examples, it is a string, but in more complicated examples (in later chapters) it can be a tuple, a frozen set, or a Python object.

In the following code, "`raise NotImplementedError()`" is a way to specify that this is an abstract method that needs to be overridden to define an actual search problem.

searchProblem.py — representations of search problems

```
11 from display import Displayable
12 import matplotlib.pyplot as plt
13 import random
14
15 class Search_problem(Displayable):
16     """A search problem consists of:
```

```
    * a start node
    * a neighbors function that gives the neighbors of a node
    * a specification of a goal
    * a (optional) heuristic function.
    The methods must be overridden to define a search problem."""

    def start_node(self):
        """returns start node"""
        raise NotImplementedError("start_node") # abstract method

    def is_goal(self,node):
        """is True if node is a goal"""
        raise NotImplementedError("is_goal") # abstract method

    def neighbors(self,node):
        """returns a list (or enumeration) of the arcs for the neighbors of
            node"""
        raise NotImplementedError("neighbors") # abstract method

    def heuristic(self,n):
        """Gives the heuristic value of node n.
        Returns 0 if not overridden."""
        return 0
```

The neighbors is a list or enumeration of arcs. A (directed) arc is the pair (`from_node`, `to_node`), but can also contain a non-negative `cost` (which defaults to 1) and can be labeled with an `action`. The action is not used for the search, but is useful for displaying and for plans (sequences of of actions).

searchProblem.py — (continued)

```
class Arc(object):
    """An arc consists of
       a from_node and a to_node node
       a (non-negative) cost
       an (optional) action
    """
    def __init__(self, from_node, to_node, cost=1, action=None):
        self.from_node = from_node
        self.to_node = to_node
        self.cost = cost
        assert cost >= 0, (f"Cost cannot be negative: {self}, cost={cost}")
        self.action = action

    def __repr__(self):
        """string representation of an arc"""
        if self.action:
            return f"{self.from_node} --{self.action}--> {self.to_node}"
        else:
            return f"{self.from_node} --> {self.to_node}"
```

3.1.1 Explicit Representation of Search Graph

The first representation of a search problem is from an explicit graph (as opposed to one that is generated as needed).

An **explicit graph** consists of

- a list or set of nodes
- a list or set of arcs
- a start node
- a list or set of goal nodes
- (optionally) a hmap dictionary that maps a node to a heuristic value for that node. This could conceivably have been part of nodes, but the heuristic value depends on the goals.
- (optionally) a positions dictionary that maps nodes to their x-y position. This is for showing the graph visually.

To define a search problem, you need to define the start node, the goal predicate, the neighbors function and, for some algorithms, a heuristic function.

searchProblem.py — (continued)

```
60 class Search_problem_from_explicit_graph(Search_problem):
61     """A search problem from an explicit graph.
62     """
63 
64     def __init__(self, title, nodes, arcs, start=None, goals=set(), hmap={},
65                  positions=None):
66         """ A search problem consists of:
67         * list or set of nodes
68         * list or set of arcs
69         * start node
70         * list or set of goal nodes
71         * hmap: dictionary that maps each node into its heuristic value.
72         * positions: dictionary that maps each node into its (x,y) position
73         """
74         self.title = title
75         self.neighs = {}
76         self.nodes = nodes
77         for node in nodes:
78             self.neighs[node]=[]
79         self.arcs = arcs
80         for arc in arcs:
81             self.neighs[arc.from_node].append(arc)
82         self.start = start
83         self.goals = goals
84         self.hmap = hmap
85         if positions is None:
```

```
86              self.positions = {node:(random.random(),random.random()) for
                    node in nodes}
87          else:
88              self.positions = positions
89
90      def start_node(self):
91          """returns start node"""
92          return self.start
93
94      def is_goal(self,node):
95          """is True if node is a goal"""
96          return node in self.goals
97
98      def neighbors(self,node):
99          """returns the neighbors of node (a list of arcs)"""
100         return self.neighs[node]
101
102     def heuristic(self,node):
103         """Gives the heuristic value of node n.
104         Returns 0 if not overridden in the hmap."""
105         if node in self.hmap:
106             return self.hmap[node]
107         else:
108             return 0
109
110     def __repr__(self):
111         """returns a string representation of the search problem"""
112         res=""
113         for arc in self.arcs:
114             res += f"{arc}. "
115         return res
```

Graphical Display of a Search Graph

The `show()` method displays the graph, and is used for the figures in this document.

searchProblem.py — (continued)

```
117     def show(self, fontsize=10, node_color='orange', show_costs = True):
118         """Show the graph as a figure
119         """
120         self.fontsize = fontsize
121         self.show_costs = show_costs
122         plt.ion()  # interactive
123         fig, ax = plt.subplots()
124         ax.set_axis_off()
125         ax.set_title(self.title, fontsize=fontsize)
126         self.show_graph(ax, node_color)
127
128     def show_graph(self, ax, node_color='orange'):
```

```
        bbox =
            dict(boxstyle="round4,pad=1.0,rounding_size=0.5",facecolor=node_color)
        for arc in self.arcs:
            self.show_arc(ax, arc)
        for node in self.nodes:
            self.show_node(ax, node, node_color = node_color)

    def show_node(self, ax, node, node_color):
            x,y = self.positions[node]
            ax.text(x,y,node,bbox=dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
                                facecolor=node_color),
                    ha='center',va='center', fontsize=self.fontsize)

    def show_arc(self, ax, arc, arc_color='black', node_color='white'):
            from_pos = self.positions[arc.from_node]
            to_pos = self.positions[arc.to_node]
            ax.annotate(arc.to_node, from_pos, xytext=to_pos,
                        arrowprops={'arrowstyle':'<|-', 'linewidth': 2,
                                    'color':arc_color},
                        bbox=dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
                                          facecolor=node_color),
                               ha='center',va='center',
                               fontsize=self.fontsize)
            # Add costs to middle of arcs:
            if self.show_costs:
                ax.text((from_pos[0]+to_pos[0])/2, (from_pos[1]+to_pos[1])/2,
                         arc.cost, bbox=dict(pad=1,fc='w',ec='w'),
                         ha='center',va='center',fontsize=self.fontsize)
```

3.1.2 Paths

A searcher will return a path from the start node to a goal node. A Python list is not a suitable representation for a path, as many search algorithms consider multiple paths at once, and these paths should share initial parts of the path. If we wanted to do this with Python lists, we would need to keep copying the list, which can be expensive if the list is long. An alternative representation is used here in terms of a recursive data structure that can share subparts.

A path is either:

- a node (representing a path of length 0) or
- an initial path, and an arc at the end, where the `from_node` of the arc is the node at the end of the initial path.

These cases are distinguished in the following code by having `arc=None` if the path has length 0, in which case `initial` is the node of the path. Note that we only use the most basic form of Python's `yield` for enumerations (Section 1.5.3).

searchProblem.py — (continued)

```
class Path(object):
    """A path is either a node or a path followed by an arc"""

    def __init__(self,initial,arc=None):
        """initial is either a node (in which case arc is None) or
        a path (in which case arc is an object of type Arc)"""
        self.initial = initial
        self.arc=arc
        if arc is None:
            self.cost=0
        else:
            self.cost = initial.cost+arc.cost

    def end(self):
        """returns the node at the end of the path"""
        if self.arc is None:
            return self.initial
        else:
            return self.arc.to_node

    def nodes(self):
        """enumerates the nodes of the path from the last element backwards
        """
        current = self
        while current.arc is not None:
            yield current.arc.to_node
            current = current.initial
        yield current.initial

    def initial_nodes(self):
        """enumerates the nodes for the path before the end node.
        This calls nodes() for the initial part of the path.
        """
        if self.arc is not None:
            yield from self.initial.nodes()

    def __repr__(self):
        """returns a string representation of a path"""
        if self.arc is None:
            return str(self.initial)
        elif self.arc.action:
            return f"{self.initial}\n --{self.arc.action}-->
                {self.arc.to_node}"
        else:
            return f"{self.initial} --> {self.arc.to_node}"
```

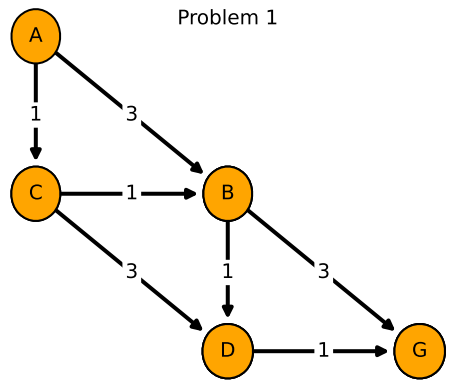


Figure 3.1: problem1

3.1.3 Example Search Problems

The first search problem is one with 5 nodes where the least-cost path is one with many arcs. See Figure 3.1, generated using problem1.show(). Note that this example is used for the unit tests, so the test (in `searchGeneric`) will need to be changed if this is changed.

searchExample.py — Search Examples

```
from searchProblem import Arc, Search_problem_from_explicit_graph,
    Search_problem

problem1 = Search_problem_from_explicit_graph('Problem 1',
    {'A','B','C','D','G'},
    [Arc('A','B',3), Arc('A','C',1), Arc('B','D',1), Arc('B','G',3),
        Arc('C','B',1), Arc('C','D',3), Arc('D','G',1)],
    start = 'A',
    goals = {'G'},
    positions={'A': (0, 1), 'B': (0.5, 0.5), 'C': (0,0.5),
                'D': (0.5,0), 'G': (1,0)})
```

The second search problem is one with 8 nodes where many paths do not lead to the goal. See Figure 3.2.

searchExample.py — (continued)

```
problem2 = Search_problem_from_explicit_graph('Problem 2',
    {'A','B','C','D','E','G','H','J'},
    [Arc('A','B',1), Arc('B','C',3), Arc('B','D',1), Arc('D','E',3),
       Arc('D','G',1), Arc('A','H',3), Arc('H','J',1)],
    start = 'A',
    goals = {'G'},
    positions={'A':(0, 1), 'B':(0, 3/4), 'C':(0,0), 'D':(1/4,3/4),
                'E':(1/4,0), 'G':(2/4,3/4), 'H':(3/4,1), 'J':(3/4,3/4)})
```

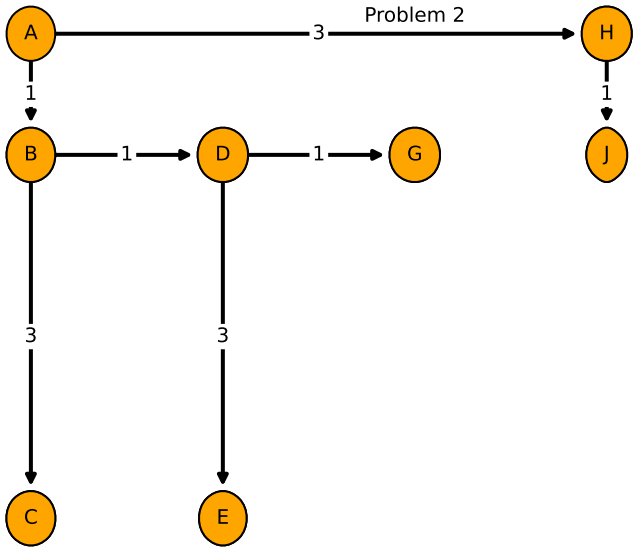


Figure 3.2: problem2

The third search problem is a disconnected graph (contains no arcs), where the start node is a goal node. This is a boundary case to make sure that weird cases work.

searchExample.py — (continued)

```
problem3 = Search_problem_from_explicit_graph('Problem 3',
    {'a','b','c','d','e','g','h','j'},
    [],
    start = 'g',
    goals = {'k','g'})
```

The `simp_delivery_graph` is shown Figure 3.3. This is the same as Figure 3.3 of Poole and Mackworth [2023].

searchExample.py — (continued)

```
simp_delivery_graph = Search_problem_from_explicit_graph("Acyclic Delivery
    Graph",
    {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J'},
    [   Arc('A', 'B', 2),
        Arc('A', 'C', 3),
        Arc('A', 'D', 4),
        Arc('B', 'E', 2),
        Arc('B', 'F', 3),
        Arc('C', 'J', 7),
        Arc('D', 'H', 4),
        Arc('F', 'D', 2),
```

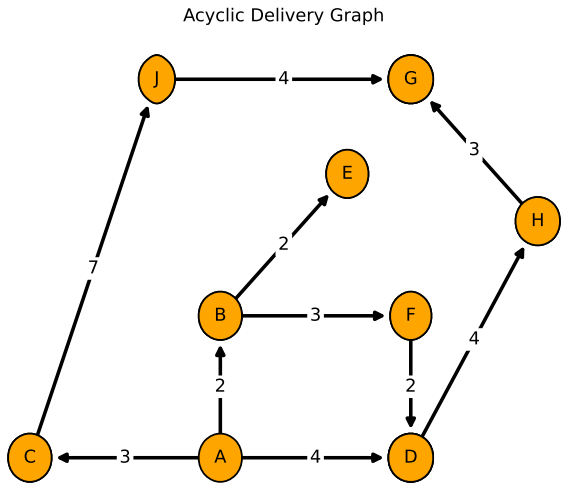


Figure 3.3: simp_delivery_graph.show()

```
47          Arc('H', 'G', 3),
48          Arc('J', 'G', 4)],
49    start = 'A',
50    goals = {'G'},
51    hmap = {
52         'A': 7,
53         'B': 5,
54         'C': 9,
55         'D': 6,
56         'E': 3,
57         'F': 5,
58         'G': 0,
59         'H': 3,
60         'J': 4,
61     },
62     positions = {
63         'A': (0.4,0.1),
64         'B': (0.4,0.4),
65         'C': (0.1,0.1),
66         'D': (0.7,0.1),
67         'E': (0.6,0.7),
68         'F': (0.7,0.4),
69         'G': (0.7,0.9),
70         'H': (0.9,0.6),
71         'J': (0.3,0.9)
72        }
```

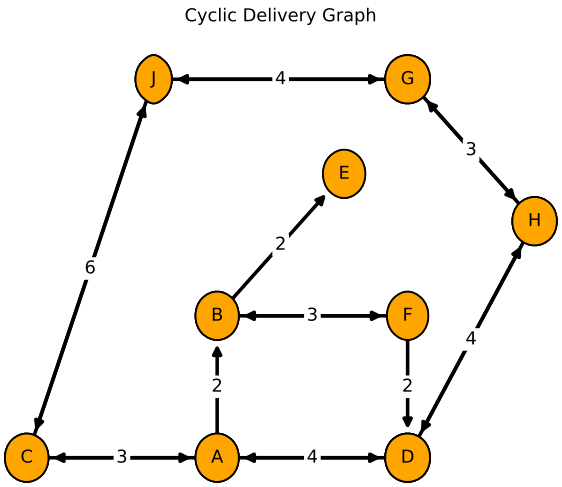


Figure 3.4: `cyclic_simp_delivery_graph.show()`

```
    )
```

`cyclic_simp_delivery_graph` is the graph shown Figure 3.4. This is the graph of Figure 3.10 of [Poole and Mackworth, 2023]. The heuristic values are the same as in `simp_delivery_graph`.

searchExample.py — (continued)

```
cyclic_simp_delivery_graph = Search_problem_from_explicit_graph("Cyclic
    Delivery Graph",
    {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J'},
    [   Arc('A', 'B', 2),
        Arc('A', 'C', 3),
        Arc('A', 'D', 4),
        Arc('B', 'E', 2),
        Arc('B', 'F', 3),
        Arc('C', 'A', 3),
        Arc('C', 'J', 6),
        Arc('D', 'A', 4),
        Arc('D', 'H', 4),
        Arc('F', 'B', 3),
        Arc('F', 'D', 2),
        Arc('G', 'H', 3),
        Arc('G', 'J', 4),
        Arc('H', 'D', 4),
        Arc('H', 'G', 3),
        Arc('J', 'C', 6),
        Arc('J', 'G', 4)],
```

```
   start = 'A',
   goals = {'G'},
   hmap = {
        'A': 7,
        'B': 5,
        'C': 9,
        'D': 6,
        'E': 3,
        'F': 5,
        'G': 0,
        'H': 3,
        'J': 4,
    },
    positions = {
        'A': (0.4,0.1),
        'B': (0.4,0.4),
        'C': (0.1,0.1),
        'D': (0.7,0.1),
        'E': (0.6,0.7),
        'F': (0.7,0.4),
        'G': (0.7,0.9),
        'H': (0.9,0.6),
        'J': (0.3,0.9)
        })
```

The next problem is the tree graph shown in Figure 3.5, and is Figure 3.15 in Poole and Mackworth [2023].

searchExample.py — (continued)

```
tree_graph = Search_problem_from_explicit_graph("Tree Graph",
   {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
       'O',
        'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'AA', 'BB',
            'CC',
        'DD', 'EE', 'FF', 'GG', 'HH', 'II', 'JJ', 'KK'},
   [    Arc('A', 'B', 1),
        Arc('A', 'C', 1),
        Arc('B', 'D', 1),
        Arc('B', 'E', 1),
        Arc('C', 'F', 1),
        Arc('C', 'G', 1),
        Arc('D', 'H', 1),
        Arc('D', 'I', 1),
        Arc('E', 'J', 1),
        Arc('E', 'K', 1),
        Arc('F', 'L', 1),
        Arc('G', 'M', 1),
        Arc('G', 'N', 1),
        Arc('H', 'O', 1),
        Arc('H', 'P', 1),
        Arc('J', 'Q', 1),
```

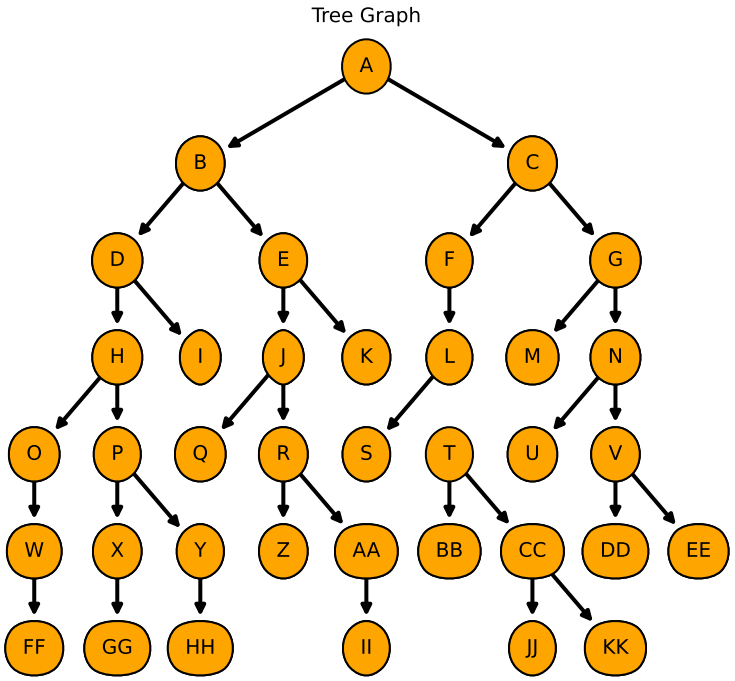


Figure 3.5: `tree_graph.show(show_costs = False)`

```
        Arc('J', 'R', 1),
        Arc('L', 'S', 1),
        Arc('L', 'T', 1),
        Arc('N', 'U', 1),
        Arc('N', 'V', 1),
        Arc('O', 'W', 1),
        Arc('P', 'X', 1),
        Arc('P', 'Y', 1),
        Arc('R', 'Z', 1),
        Arc('R', 'AA', 1),
        Arc('T', 'BB', 1),
        Arc('T', 'CC', 1),
        Arc('V', 'DD', 1),
        Arc('V', 'EE', 1),
        Arc('W', 'FF', 1),
        Arc('X', 'GG', 1),
        Arc('Y', 'HH', 1),
        Arc('AA', 'II', 1),
```

```
        Arc('CC', 'JJ', 1),
        Arc('CC', 'KK', 1)
    ],
   start = 'A',
   goals = {'K', 'M', 'T', 'X', 'Z', 'HH'},
    positions = {
        'A': (0.5,0.95),
        'B': (0.3,0.8),
        'C': (0.7,0.8),
        'D': (0.2,0.65),
        'E': (0.4,0.65),
        'F': (0.6,0.65),
        'G': (0.8,0.65),
        'H': (0.2,0.5),
        'I': (0.3,0.5),
        'J': (0.4,0.5),
        'K': (0.5,0.5),
        'L': (0.6,0.5),
        'M': (0.7,0.5),
        'N': (0.8,0.5),
        'O': (0.1,0.35),
        'P': (0.2,0.35),
        'Q': (0.3,0.35),
        'R': (0.4,0.35),
        'S': (0.5,0.35),
        'T': (0.6,0.35),
        'U': (0.7,0.35),
        'V': (0.8,0.35),
        'W': (0.1,0.2),
        'X': (0.2,0.2),
        'Y': (0.3,0.2),
        'Z': (0.4,0.2),
        'AA': (0.5,0.2),
        'BB': (0.6,0.2),
        'CC': (0.7,0.2),
        'DD': (0.8,0.2),
        'EE': (0.9,0.2),
        'FF': (0.1,0.05),
        'GG': (0.2,0.05),
        'HH': (0.3,0.05),
        'II': (0.5,0.05),
        'JJ': (0.7,0.05),
        'KK': (0.8,0.05)
        }
    )

# tree_graph.show(show_costs = False)
```

3.2 Generic Searcher and Variants

To run the search demos, in folder "aipython", load "searchGeneric.py" , using e.g., ipython -i searchGeneric.py, and copy and paste the example queries at the bottom of that file.

3.2.1 Searcher

A *Searcher* for a problem can be asked repeatedly for the next path. To solve a search problem, construct a *Searcher* object for the problem and then repeatedly ask for the next path using *search*. If there are no more paths, *None* is returned.

searchGeneric.py — Generic Searcher, including depth-first and A*

```
11 from display import Displayable
12
13 class Searcher(Displayable):
14     """returns a searcher for a problem.
15     Paths can be found by repeatedly calling search().
16     This does depth-first search unless overridden
17     """
18     def __init__(self, problem):
19         """creates a searcher from a problem
20         """
21         self.problem = problem
22         self.initialize_frontier()
23         self.num_expanded = 0
24         self.add_to_frontier(Path(problem.start_node()))
25         super().__init__()
26
27     def initialize_frontier(self):
28         self.frontier = []
29
30     def empty_frontier(self):
31         return self.frontier == []
32
33     def add_to_frontier(self,path):
34         self.frontier.append(path)
35
36     def search(self):
37         """returns (next) path from the problem's start node
38         to a goal node.
39         Returns None if no path exists.
40         """
41         while not self.empty_frontier():
42             self.path = self.frontier.pop()
43             self.num_expanded += 1
44             if self.problem.is_goal(self.path.end()): # solution found
45                 self.solution = self.path # store the solution found
```

```
46              self.display(1, f"Solution: {self.path} (cost:
                    {self.path.cost})\n",
47                  self.num_expanded, "paths have been expanded and",
48                         len(self.frontier), "paths remain in the
                               frontier")
49              return self.path
50          else:
51              self.display(4,f"Expanding: {self.path} (cost:
                    {self.path.cost})")
52              neighs = self.problem.neighbors(self.path.end())
53              self.display(2,f"Expanding: {self.path} with neighbors
                    {neighs}")
54              for arc in reversed(list(neighs)):
55                  self.add_to_frontier(Path(self.path,arc))
56              self.display(3, f"New frontier: {[p.end() for p in
                    self.frontier]}")
57
58      self.display(0,"No (more) solutions. Total of",
59                   self.num_expanded,"paths expanded.")
```

Note that this reverses the neighbors so that it implements depth-first search in an intuitive manner (expanding the first neighbor first). The call to *list* is for the case when the neighbors are generated (and not already in a list). Reversing the neighbors might not be required for other methods. The calls to *reversed* and *list* can be removed, and the algorithm still implements depth-first search.

To use depth-first search to find multiple paths for `problem1` and `simp_delivery_graph`, copy and paste the following into Python's read-evaluate-print loop; keep finding next solutions until there are no more:

searchGeneric.py — (continued)

```
61 # Depth-first search for problem1:
62 # searcher1 = Searcher(searchExample.problem1)
63 # searcher1.search() # find first solution
64 # searcher1.search() # find next solution (repeat until no solutions)
65
66 # Depth-first search for simple delivery graph:
67 # searcher_sdg = Searcher(searchExample.simp_delivery_graph)
68 # searcher_sdg.search() # find first or next solution
```

Exercise 3.1 Implement breadth-first search. Only *add_to_frontier* and/or *pop* need to be modified to implement a first-in first-out queue.

3.2.2 GUI for Tracing Search

[This GUI implements most of the functionality of the solve model of the now-discontinued AISpace.org search app.]

Figure 3.6 shows the GUI that can be used to step through search algorithms. Here the path $A \rightarrow B$ is being expanded, and the neighbors are E and F. The other nodes at the end of paths of the frontier are C and D. Thus the

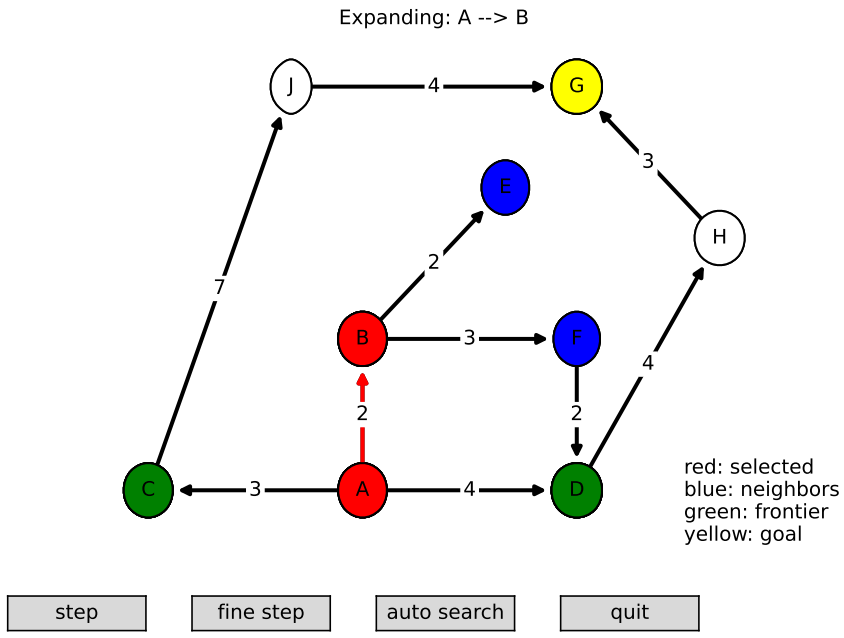


Figure 3.6: `SearcherGUI(Searcher, simp_delivery_graph)`

frontier contains paths to C and D, used to also contain $A \rightarrow B$, and now will contain $A \rightarrow B \rightarrow E$ and $A \rightarrow B \rightarrow F$.

`SearcherGUI` takes a search class and a problem, and lets one explore the search space after calling `go()`. A GUI can only be used for one search; at the end of the search the loop ends and the buttons no longer work.

This is implemented by redefining `display`. The search algorithms don't need to be modified. If you modify them (or create your own), you just have to be careful to use the appropriate number for the display. The first argument to display has the following meanings:

1. a solution has been found
2. what is shown for a "step" on a GUI; here it is assumed to be the path, the neighbors of the end of the path, and the other nodes at the end of paths on the frontier
3. (shown with "fine step" but not with "step") the frontier and the path selected
4. (shown with "fine step" but not with "step") the frontier.

It is also useful to look at the Python console, as the display information is printed there.

searchGUI.py — GUI for search

```
import matplotlib.pyplot as plt
from matplotlib.widgets import Button
import time

class SearcherGUI(object):
    def __init__(self, SearchClass, problem,
                 fontsize=10,
                 colors = {'selected':'red', 'neighbors':'blue',
                     'frontier':'green', 'goal':'yellow'},
                 show_costs = True):
        self.problem = problem
        self.searcher = SearchClass(problem)
        self.problem.fontsize = fontsize
        self.colors = colors
        self.problem.show_costs = show_costs
        self.quitting = False

        fig, self.ax = plt.subplots()
        plt.ion()  # interactive
        self.ax.set_axis_off()
        plt.subplots_adjust(bottom=0.15)
        step_butt = Button(fig.add_axes([0.1,0.02,0.2,0.05]), "step")
        step_butt.on_clicked(self.step)
        fine_butt = Button(fig.add_axes([0.4,0.02,0.2,0.05]), "fine step")
        fine_butt.on_clicked(self.finestep)
        auto_butt = Button(fig.add_axes([0.7,0.02,0.2,0.05]), "auto search")
        auto_butt.on_clicked(self.auto)
        fig.canvas.mpl_connect('close_event', self.window_closed)
        self.ax.text(0.85,0, '\n'.join(self.colors[a]+": "+a
                                       for a in self.colors))
        self.problem.show_graph(self.ax, node_color='white')
        self.problem.show_node(self.ax, self.problem.start,
                               self.colors['frontier'])
        for node in self.problem.nodes:
            if self.problem.is_goal(node):
                self.problem.show_node(self.ax, node,self.colors['goal'])
        plt.show()
        self.click = 7 # bigger than any display!
        self.searcher.display = self.display
        try:
            while self.searcher.frontier:
                path = self.searcher.search()
        except ExitToPython:
            print("GUI closed")
        else:
            print("No more solutions")

    def display(self, level, *args, **nargs):
        if self.quitting:
```

```
            raise ExitToPython()
        if level <= self.click: #step
            print(*args, **nargs)
            self.ax.set_title(f"Expanding: {self.searcher.path}",
                              fontsize=self.problem.fontsize)
            if level == 1:
                self.show_frontier(self.colors['frontier'])
                self.show_path(self.colors['selected'])
                self.ax.set_title(f"Solution Found: {self.searcher.path}",
                                  fontsize=self.problem.fontsize)
            elif level == 2: # what should be shown if node in multiple?
                self.show_frontier(self.colors['frontier'])
                self.show_path(self.colors['selected'])
                self.show_neighbors(self.colors['neighbors'])
            elif level == 3:
                self.show_frontier(self.colors['frontier'])
                self.show_path(self.colors['selected'])
            elif level == 4:
                self.show_frontier(self.colors['frontier'])


            # wait for a button click
            self.click = 0
            plt.draw()
            while self.click == 0 and not self.quitting:
                plt.pause(0.1)
            if self.quitting:
                raise ExitToPython()
            # undo coloring:
            self.ax.set_title("")
            self.show_frontier('white')
            self.show_neighbors('white')
            path_show = self.searcher.path
            while path_show.arc:
                self.problem.show_arc(self.ax, path_show.arc, 'black')
                self.problem.show_node(self.ax, path_show.end(), 'white')
                path_show = path_show.initial
            self.problem.show_node(self.ax, path_show.end(), 'white')
            if self.problem.is_goal(self.searcher.path.end()):
                self.problem.show_node(self.ax, self.searcher.path.end(),
                                       self.colors['goal'])
            plt.draw()

    def show_frontier(self, color):
        for path in self.searcher.frontier:
            self.problem.show_node(self.ax, path.end(), color)

    def show_path(self, color):
        """color selected path"""
        path_show = self.searcher.path
```

```
        while path_show.arc:
                self.problem.show_arc(self.ax, path_show.arc, color)
                self.problem.show_node(self.ax, path_show.end(), color)
                path_show = path_show.initial
        self.problem.show_node(self.ax, path_show.end(), color)

    def show_neighbors(self, color):
        for neigh in self.problem.neighbors(self.searcher.path.end()):
            self.problem.show_node(self.ax, neigh.to_node, color)

    def auto(self, event):
        self.click = 1
    def step(self,event):
        self.click = 2
    def finestep(self, event):
        self.click = 3
    def window_closed(self, event):
        self.quitting = True

class ExitToPython(Exception):
    pass
```

searchGUI.py — (continued)

```
from searchGeneric import Searcher, AStarSearcher
from searchMPP import SearcherMPP
import searchExample
from searchBranchAndBound import DF_branch_and_bound

# to demonstrate depth-first search:
# sdfs = SearcherGUI(Searcher, searchExample.tree_graph)

# delivery graph examples:
# sh = SearcherGUI(Searcher, searchExample.simp_delivery_graph)
# sha = SearcherGUI(AStarSearcher, searchExample.simp_delivery_graph)
# shac = SearcherGUI(AStarSearcher,
    searchExample.cyclic_simp_delivery_graph)
# shm = SearcherGUI(SearcherMPP, searchExample.cyclic_simp_delivery_graph)
# shb = SearcherGUI(DF_branch_and_bound, searchExample.simp_delivery_graph)

# The following is AI:FCA figure 3.15, and is useful to show branch&bound:
# shbt = SearcherGUI(DF_branch_and_bound, searchExample.tree_graph)

if __name__ == "__main__":
    print("Try e.g.: SearcherGUI(Searcher,
        searchExample.simp_delivery_graph)")
```

3.2.3 Frontier as a Priority Queue

In many of the search algorithms, such as A^* and other best-first searchers, the frontier is implemented as a priority queue. The following code uses the Python's built-in priority queue implementations, `heapq`.

Following the lead of the Python documentation, `https://docs.python.org/3/library/heapq.html`, a frontier is a list of triples. The first element of each triple is the value to be minimized. The second element is a unique index which specifies the order that the elements were added to the queue, and the third element is the path that is on the queue. The use of the unique index ensures that the priority queue implementation does not compare paths; whether one path is less than another is not defined. It also lets us control what sort of search (e.g., depth-first or breadth-first) occurs when the value to be minimized does not give a unique next path.

The variable *frontier_index* is the total number of elements of the frontier that have been created. As well as being used as the unique index, it is useful for statistics, particularly in conjunction with the current size of the frontier.

searchGeneric.py — (continued)

```
70 import heapq      # part of the Python standard library
71 from searchProblem import Path
72
73 class FrontierPQ(object):
74     """A frontier consists of a priority queue (heap), frontierpq, of
75         (value, index, path) triples, where
76     * value is the value we want to minimize (e.g., path cost + h).
77     * index is a unique index for each element
78     * path is the path on the queue
79     Note that the priority queue always returns the smallest element.
80     """
81
82     def __init__(self):
83         """constructs the frontier, initially an empty priority queue
84         """
85         self.frontier_index = 0 # the number of items added to the frontier
86         self.frontierpq = [] # the frontier priority queue
87
88     def empty(self):
89         """is True if the priority queue is empty"""
90         return self.frontierpq == []
91
92     def add(self, path, value):
93         """add a path to the priority queue
94         value is the value to be minimized"""
95         self.frontier_index += 1 # get a new unique index
96         heapq.heappush(self.frontierpq,(value, -self.frontier_index, path))
97
98     def pop(self):
99         """returns and removes the path of the frontier with minimum value.
```

```
        """
        (_,_,path) = heapq.heappop(self.frontierpq)
        return path
```

The following methods are used for finding and printing information about the frontier.

searchGeneric.py — (continued)

```
    def count(self,val):
        """returns the number of elements of the frontier with value=val"""
        return sum(1 for e in self.frontierpq if e[0]==val)

    def __repr__(self):
        """string representation of the frontier"""
        return str([(n,c,str(p)) for (n,c,p) in self.frontierpq])

    def __len__(self):
        """length of the frontier"""
        return len(self.frontierpq)

    def __iter__(self):
        """iterate through the paths in the frontier"""
        for (_,_,path) in self.frontierpq:
            yield path
```

3.2.4 A^* Search

For an A^* **Search** the frontier is implemented using the FrontierPQ class.

searchGeneric.py — (continued)

```
class AStarSearcher(Searcher):
    """returns a searcher for a problem.
    Paths can be found by repeatedly calling search().
    """

    def __init__(self, problem):
        super().__init__(problem)

    def initialize_frontier(self):
        self.frontier = FrontierPQ()

    def empty_frontier(self):
        return self.frontier.empty()

    def add_to_frontier(self,path):
        """add path to the frontier with the appropriate cost"""
        value = path.cost+self.problem.heuristic(path.end())
        self.frontier.add(path, value)
```

Code should always be tested. The following provides a simple **unit test**, using `problem1` as the default problem.

searchGeneric.py — (continued)

```
140 import searchExample
141
142 def test(SearchClass, problem=searchExample.problem1,
        solutions=[['G','D','B','C','A']] ):
143     """Unit test for aipython searching algorithms.
144     SearchClass is a class that takes a problem and implements search()
145     problem is a search problem
146     solutions is a list of optimal solutions
147     """
148     print("Testing problem 1:")
149     schr1 = SearchClass(problem)
150     path1 = schr1.search()
151     print("Path found:",path1)
152     assert path1 is not None, "No path is found in problem1"
153     assert list(path1.nodes()) in solutions, "Shortest path not found in
            problem1"
154     print("Passed unit test")
155
156 if __name__ == "__main__":
157     #test(Searcher)    # what needs to be changed to make this succeed?
158     test(AStarSearcher)
159
160 # example queries:
161 # searcher1 = Searcher(searchExample.simp_delivery_graph) # DFS
162 # searcher1.search() # find first path
163 # searcher1.search() # find next path
164 # searcher2 = AStarSearcher(searchExample.simp_delivery_graph) # A*
165 # searcher2.search() # find first path
166 # searcher2.search() # find next path
167 # searcher3 = Searcher(searchExample.cyclic_simp_delivery_graph) # DFS
168 # searcher3.search() # find first path with DFS. What do you expect to
        happen?
169 # searcher4 = AStarSearcher(searchExample.cyclic_simp_delivery_graph) # A*
170 # searcher4.search() # find first path
171
172 # To use the GUI for A* search do the following
173 # python -i searchGUI.py
174 # SearcherGUI(AStarSearcher, searchExample.simp_delivery_graph)
175 # SearcherGUI(AStarSearcher, searchExample.cyclic_simp_delivery_graph)
```

Exercise 3.2 Change the code so that it implements (i) best-first search and (ii) lowest-cost-first search. For each of these methods compare it to A^* in terms of the number of paths expanded, and the path found.

Exercise 3.3 The searcher acts like a Python iterator, in that it returns one value (here a path) and then returns other values (paths) on demand, but does not implement the iterator interface. Change the code so it implements the iterator interface. What does this enable us to do?

3.2.5 Multiple Path Pruning

> To run the multiple-path pruning demo, in folder "aipython", load "searchMPP.py" , using e.g., ipython -i searchMPP.py, and copy and paste the example queries at the bottom of that file.

The following implements A^* with multiple-path pruning. It overrides *search*() in *Searcher*.

searchMPP.py — Searcher with multiple-path pruning

```
from searchGeneric import AStarSearcher
from searchProblem import Path

class SearcherMPP(AStarSearcher):
    """returns a searcher for a problem.
    Paths can be found by repeatedly calling search().
    """
    def __init__(self, problem):
        super().__init__(problem)
        self.explored = set()

    def search(self):
        """returns next path from an element of problem's start nodes
        to a goal node.
        Returns None if no path exists.
        """
        while not self.empty_frontier():
            self.path = self.frontier.pop()
            if self.path.end() not in self.explored:
                self.explored.add(self.path.end())
                self.num_expanded += 1
                if self.problem.is_goal(self.path.end()):
                    self.solution = self.path # store the solution found
                    self.display(1, f"Solution: {self.path} (cost:
                        {self.path.cost})\n",
                    self.num_expanded, "paths have been expanded and",
                           len(self.frontier), "paths remain in the
                               frontier")
                    return self.path
                else:
                    self.display(4,f"Expanding: {self.path} (cost:
                        {self.path.cost})")
                    neighs = self.problem.neighbors(self.path.end())
                    self.display(2,f"Expanding: {self.path} with neighbors
                        {neighs}")
                    for arc in neighs:
                        self.add_to_frontier(Path(self.path,arc))
                    self.display(3, f"New frontier: {[p.end() for p in
                        self.frontier]}")
        self.display(0,"No (more) solutions. Total of",
```

```
                    self.num_expanded,"paths expanded.")

from searchGeneric import test
if __name__ == "__main__":
    test(SearcherMPP)

import searchExample
# searcherMPPcdp = SearcherMPP(searchExample.cyclic_simp_delivery_graph)
# searcherMPPcdp.search() # find first path

# To use the GUI for SearcherMPP do
# python -i searchGUI.py
# import searchMPP
# SearcherGUI(searchMPP.SearcherMPP,
    searchExample.cyclic_simp_delivery_graph)
```

Exercise 3.4 Chris was very puzzled as to why there was a minus ("$-$") in the second element of the tuple added to the heap in the `add` method in `FrontierPQ` in `searchGeneric.py`.

Sam suggested the following example would demonstrate the importance of the minus. Consider an infinite integer grid, where the states are pairs of integers, the start is (0,0), and the goal is (10,10). The neighbors of (i, j) are $(i+1, j)$ and $(i, j+1)$. Consider the heuristic function $h((i, j)) = |10-i|+|10-j|$. Sam suggested you compare how many paths are expanded with the minus and without the minus. `searchGrid` is a representation of Sam's graph. If something takes too long, you might consider changing the size.

searchGrid.py — A grid problem to demonstrate A*

```
from searchProblem import Search_problem, Arc

class GridProblem(Search_problem):
    """a node is a pair (x,y)"""
    def __init__(self, size=10):
        self.size = size

    def start_node(self):
        """returns the start node"""
        return (0,0)

    def is_goal(self,node):
        """returns True when node is a goal node"""
        return node == (self.size,self.size)

    def neighbors(self,node):
        """returns a list of the neighbors of node"""
        (x,y) = node
        return [Arc(node,(x+1,y)), Arc(node,(x,y+1))]

    def heuristic(self,node):
        (x,y) = node
```

```
33          return abs(x-self.size)+abs(y-self.size)
34
35  class GridProblemNH(GridProblem):
36      """Grid problem with a heuristic of 0"""
37      def heuristic(self,node):
38          return 0
39
40  from searchGeneric import Searcher, AStarSearcher
41  from searchMPP import SearcherMPP
42  from searchBranchAndBound import DF_branch_and_bound
43
44  def testGrid(size = 10):
45      print("\nWith MPP")
46      gridsearchermpp = SearcherMPP(GridProblem(size))
47      print(gridsearchermpp.search())
48      print("\nWithout MPP")
49      gridsearchera = AStarSearcher(GridProblem(size))
50      print(gridsearchera.search())
51      print("\nWith MPP and a heuristic = 0 (Dijkstra's algorithm)")
52      gridsearchermppnh = SearcherMPP(GridProblemNH(size))
53      print(gridsearchermppnh.search())
```

Explain to Chris what the minus does and why it is there. Give evidence for your claims. It might be useful to refer to other search strategies in your explanation. As part of your explanation, explain what is special about Sam's example.

Exercise 3.5 Implement a searcher that implements cycle pruning instead of multiple-path pruning. You need to decide whether to check for cycles when paths are added to the frontier or when they are removed. (Hint: either method can be implemented by only changing one or two lines in SearcherMPP. Hint: there is a cycle if `path.end() in path.initial_nodes()`) Compare no pruning, multiple path pruning and cycle pruning for the cyclic delivery problem. Which works better in terms of number of paths expanded, computational time or space?

3.3 Branch-and-bound Search

To run the demo, in folder "`aipython`", load "`searchBranchAndBound.py`", and copy and paste the example queries at the bottom of that file.

Depth-first search methods do not need a priority queue, but can use a list as a stack. In this implementation of branch-and-bound search, we call *search* to find an optimal solution with cost less than bound. This uses depth-first search to find a path to a goal that extends *path* with cost less than the bound. Once a path to a goal has been found, that path is remembered as the *best_path*, the bound is reduced, and the search continues.

searchBranchAndBound.py — Branch and Bound Search

```
11  from searchProblem import Path
```

```
from searchGeneric import Searcher
from display import Displayable

class DF_branch_and_bound(Searcher):
    """returns a branch and bound searcher for a problem.
    An optimal path with cost less than bound can be found by calling search()
    """
    def __init__(self, problem, bound=float("inf")):
        """creates a searcher than can be used with search() to find an optimal path.
        bound gives the initial bound. By default this is infinite - meaning there
        is no initial pruning due to depth bound
        """
        super().__init__(problem)
        self.best_path = None
        self.bound = bound

    def search(self):
        """returns an optimal solution to a problem with cost less than bound.
        returns None if there is no solution with cost less than bound."""
        self.frontier = [Path(self.problem.start_node())]
        self.num_expanded = 0
        while self.frontier:
            self.path = self.frontier.pop()
            if self.path.cost+self.problem.heuristic(self.path.end()) < self.bound:
                # if self.path.end() not in self.path.initial_nodes(): # for cycle pruning
                self.display(2,"Expanding:",self.path,"cost:",self.path.cost)
                self.num_expanded += 1
                if self.problem.is_goal(self.path.end()):
                    self.best_path = self.path
                    self.bound = self.path.cost
                    self.display(1,"New best path:",self.path,"cost:",self.path.cost)
                else:
                    neighs = self.problem.neighbors(self.path.end())
                    self.display(4,"Neighbors are", neighs)
                    for arc in reversed(list(neighs)):
                        self.add_to_frontier(Path(self.path, arc))
                    self.display(3, f"New frontier: {[p.end() for p in self.frontier]}")
        self.path = self.best_path
        self.solution = self.best_path
        self.display(1,f"Optimal solution is {self.best_path}." if self.best_path
                       else "No solution found.",
```

```
                        f"Number of paths expanded: {self.num_expanded}.")
        return self.best_path
```

Note that this code used *reversed* in order to expand the neighbors of a node in the left-to-right order one might expect. It does this because *pop*() removes the rightmost element of the list. The call to *list* is there because *reversed* only works on lists and tuples, but the neighbors can be generated.

Here is a unit test and some queries:

searchBranchAndBound.py — (continued)

```
from searchGeneric import test
if __name__ == "__main__":
    test(DF_branch_and_bound)

# Example queries:
import searchExample
# searcherb1 = DF_branch_and_bound(searchExample.simp_delivery_graph)
# searcherb1.search()     # find optimal path
# searcherb2 =
    DF_branch_and_bound(searchExample.cyclic_simp_delivery_graph,
    bound=100)
# searcherb2.search()     # find optimal path

# to use the GUI do:
# ipython -i searchGUI.py
# import searchBranchAndBound
# SearcherGUI(searchBranchAndBound.DF_branch_and_bound,
    searchExample.simp_delivery_graph)
# SearcherGUI(searchBranchAndBound.DF_branch_and_bound,
    searchExample.cyclic_simp_delivery_graph)
```

Exercise 3.6 In searcherb2, in the code above, what happens if the bound is smaller, say 10? What if it is larger, say 1000?

Exercise 3.7 Implement a branch-and-bound search using recursion. Hint: you don't need an explicit frontier, but can do a recursive call for the children.

Exercise 3.8 Add loop detection to branch-and-bound search.

Exercise 3.9 After the branch-and-bound search found a solution, Sam ran search again, and noticed a different count. Sam hypothesized that this count was related to the number of nodes that an $A*$ search would use (either expand or be added to the frontier). Or maybe, Sam thought, the count for a number of nodes when the bound is slightly above the optimal path case is related to how A^* would work. Is there a relationship between these counts? Are there different things that it could count so they are related? Try to find the most specific statement that is true, and explain why it is true.

To test the hypothesis, Sam wrote the following code, but isn't sure it is helpful:

searchTest.py — code that may be useful to compare A* and branch-and-bound

```
from searchGeneric import Searcher, AStarSearcher
```

```
from searchBranchAndBound import DF_branch_and_bound
from searchMPP import SearcherMPP

DF_branch_and_bound.max_display_level = 1
Searcher.max_display_level = 1

def run(problem,name):
    print("\n\n*******",name)

    print("\nA*:")
    asearcher = AStarSearcher(problem)
    print("Path found:",asearcher.search()," cost=",asearcher.solution.cost)
    print("there are",asearcher.frontier.count(asearcher.solution.cost),
          "elements remaining on the queue with
              f-value=",asearcher.solution.cost)

    print("\nA* with MPP:"),
    msearcher = SearcherMPP(problem)
    print("Path found:",msearcher.search()," cost=",msearcher.solution.cost)
    print("there are",msearcher.frontier.count(msearcher.solution.cost),
          "elements remaining on the queue with
              f-value=",msearcher.solution.cost)

    bound = asearcher.solution.cost*1.00001
    print("\nBranch and bound (with too-good initial bound of", bound,")")
    tbb = DF_branch_and_bound(problem,bound) # cheating!!!!
    print("Path found:",tbb.search()," cost=",tbb.solution.cost)
    print("Rerunning B&B")
    print("Path found:",tbb.search())

    bbound = asearcher.solution.cost*10+10
    print("\nBranch and bound (with not-very-good initial bound of",
        bbound, ")")
    tbb2 = DF_branch_and_bound(problem,bbound)
    print("Path found:",tbb2.search()," cost=",tbb2.solution.cost)
    print("Rerunning B&B")
    print("Path found:",tbb2.search())

    print("\nDepth-first search: (Use ^C if it goes on forever)")
    tsearcher = Searcher(problem)
    print("Path found:",tsearcher.search()," cost=",tsearcher.solution.cost)


import searchExample
from searchTest import run
if __name__ == "__main__":
    run(searchExample.problem1,"Problem 1")
#   run(searchExample.simp_delivery_graph,"Acyclic Delivery")
#   run(searchExample.cyclic_simp_delivery_graph,"Cyclic Delivery")
# also test graphs with cycles, and graphs with multiple least-cost paths
```

Chapter 4

Reasoning with Constraints

4.1 Constraint Satisfaction Problems

4.1.1 Variables

A **variable** consists of a name, a domain and an optional (x,y) position (for displaying). The domain of a variable is a list or a tuple, as the ordering matters for some algorithms.

variable.py — Representations of a variable in CSPs and probabilistic models

```
import random

class Variable(object):
    """A random variable.
    name (string) - name of the variable
    domain (list) - a list of the values for the variable.
    an (x,y) position for displaying
    """

    def __init__(self, name, domain, position=None):
        """Variable
        name a string
        domain a list of printable values
        position of form (x,y) where 0 <= x <= 1, 0 <= y <= 1
        """
        self.name = name # string
        self.domain = domain # list of values
        self.position = position if position else (random.random(),
            random.random())
        self.size = len(domain)

    def __str__(self):
```

```
32          return self.name
33
34      def __repr__(self):
35          return self.name # f"Variable({self.name})"
```

4.1.2 Constraints

A **constraint** consists of:

- A tuple (or list) of variables called the **scope**.
- A **condition**, a Boolean function that takes the same number of arguments as there are variables in the scope.
- An name (for displaying)
- An optional (x, y) position. The mean of the positions of the variables in the scope is used, if not specified.

cspProblem.py — Representations of a Constraint Satisfaction Problem

```
11 from variable import Variable
12
13 # for showing csps:
14 import matplotlib.pyplot as plt
15 import matplotlib.lines as lines
16
17 class Constraint(object):
18     """A Constraint consists of
19     * scope: a tuple or list of variables
20     * condition: a Boolean function that can applied to a tuple of values
           for variables in scope
21     * string: a string for printing the constraint
22     """
23     def __init__(self, scope, condition, string=None, position=None):
24         self.scope = scope
25         self.condition = condition
26         self.string = string
27         self.position = position
28
29     def __repr__(self):
30         return self.string
```

An **assignment** is a *variable*:*value* dictionary.
If con is a constraint:

- `con.can_evaluate(assignment)` is True when the constraint can be evaluated in the assignment. Generally this is true when all variables in the scope of the constraint are assigned in the assignment. [There are cases where it could be true when not all variables are assigned, such as if the constraint was "if x then y else z", but that it not implemented here.]

- `con.holds(assignment)` returns True or False depending on whether the condition is true or false for that assignment. The assignment `assignment` must assign a value to every variable in the scope of the constraint `con` (and could also assign values to other variables); `con.holds` gives an error if not all variables in the scope of `con` are assigned in the assignment. It ignores variables in `assignment` that are not in the scope of the constraint.

In Python, the $*$ notation is used for unpacking a tuple. For example, $F(*(1,2,3))$ is the same as $F(1,2,3)$. So if t has value $(1,2,3)$, then $F(*t)$ is the same as $F(1,2,3)$.

cspProblem.py — (continued)

```
32     def can_evaluate(self, assignment):
33         """
34         assignment is a variable:value dictionary
35         returns True if the constraint can be evaluated given assignment
36         """
37         return all(v in assignment for v in self.scope)
38 
39     def holds(self,assignment):
40         """returns the value of Constraint con evaluated in assignment.
41 
42         precondition: all variables are assigned in assignment, ie
               self.can_evaluate(assignment) is true
43         """
44         return self.condition(*tuple(assignment[v] for v in self.scope))
```

4.1.3 CSPs

A constraint satisfaction problem (CSP) requires:

- `title`: a string title
- `variables`: a list or set of variables
- `constraints`: a set or list of constraints.

Other properties are inferred from these:

- `var_to_const` is a mapping from variables to set of constraints, such that `var_to_const[var]` is the set of constraints with `var` in their scope.

cspProblem.py — (continued)

```
46 class CSP(object):
47     """A CSP consists of
48     * a title (a string)
49     * variables, a list or set of variables
50     * constraints, a list of constraints
51     * var_to_const, a variable to set of constraints dictionary
```

```
    """
    def __init__(self, title, variables, constraints):
        """title is a string
        variables is set of variables
        constraints is a list of constraints
        """
        self.title = title
        self.variables = variables
        self.constraints = constraints
        self.var_to_const = {var:set() for var in self.variables}
        for con in constraints:
            for var in con.scope:
                self.var_to_const[var].add(con)

    def __str__(self):
        """string representation of CSP"""
        return self.title

    def __repr__(self):
        """more detailed string representation of CSP"""
        return f"CSP({self.title}, {self.variables}, {([str(c) for c in
            self.constraints])})"
```

`csp.consistent(assignment)` returns true if the assignment is consistent with each of the constraints in `csp` (i.e., all of the constraints that can be evaluated evaluate to true). Unless the assignment assigns to all variables, `consistent` does *not* imply the CSP is consistent or has a solution, because constraints involving variables not in the assignment are ignored.

cspProblem.py — (continued)

```
    def consistent(self,assignment):
        """assignment is a variable:value dictionary
        returns True if all of the constraints that can be evaluated
                        evaluate to True given assignment.
        """
        return all(con.holds(assignment)
                   for con in self.constraints
                   if con.can_evaluate(assignment))
```

The **show** method uses matplotlib to show the graphical structure of a constraint network. This also includes code used for the consistency GUI (Section 4.4.2).

cspProblem.py — (continued)

```
    def show(self, linewidth=3, showDomains=False, showAutoAC = False):
        self.linewidth = linewidth
        self.picked = None
        plt.ion()  # interactive
        self.arcs = {} # arc: (con,var) dictionary
        self.thelines = {} # (con,var):arc dictionary
        self.nodes = {} # node: variable dictionary
```

```
        self.fig, self.ax= plt.subplots(1, 1)
        self.ax.set_axis_off()
        for var in self.variables:
            if var.position is None:
                var.position = (random.random(), random.random())
        self.showAutoAC = showAutoAC # used for consistency GUI
        self.autoAC = False
        domains = {var:var.domain for var in self.variables} if showDomains
            else {}
        self.draw_graph(domains=domains)

    def draw_graph(self, domains={}, to_do = {}, title=None, fontsize=10):
        self.ax.clear()
        self.ax.set_axis_off()
        if title:
            self.ax.set_title(title, fontsize=fontsize)
        else:
            self.ax.set_title(self.title, fontsize=fontsize)
        var_bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
                        facecolor="yellow")
        con_bbox = dict(boxstyle="square,pad=1.0",facecolor="lightyellow")
        self.autoACtext = self.ax.text(0,0,"Auto AC" if self.showAutoAC
            else "",
                                       bbox={'boxstyle':'square,pad=1.0',
                                             'facecolor':'pink'},
                                       picker=True, fontsize=fontsize)
        for con in self.constraints:
            if con.position is None:
                con.position = tuple(sum(var.position[i] for var in
                    con.scope)/len(con.scope)
                                     for i in range(2))
            cx,cy = con.position
            bbox = con_bbox
            for var in con.scope:
                vx,vy = var.position
                if (var,con) in to_do:
                    color = 'blue'
                else:
                    color = 'green'
                line = lines.Line2D([cx,vx], [cy,vy], axes=self.ax,
                    color=color,
                                    picker=True, pickradius=10,
                                        linewidth=self.linewidth)
                self.arcs[line]= (var,con)
                self.thelines[(var,con)] = line
                self.ax.add_line(line)
            self.ax.text(cx,cy,con.string,
                                 bbox=con_bbox,
                                 ha='center',va='center', fontsize=fontsize)
        for var in self.variables:
```

```
            x,y = var.position
            if domains:
                node_label = f"{var.name}\n{domains[var]}"
            else:
                node_label = var.name
            node = self.ax.text(x, y, node_label, bbox=var_bbox,
                ha='center', va='center',
                        picker=True, fontsize=fontsize)
            self.nodes[node] = var
        self.fig.canvas.mpl_connect('pick_event', self.pick_handler)
```

The following method is used for the GUI (Section 4.4.2).

cspProblem.py — (continued)

```
    def pick_handler(self,event):
        mouseevent = event.mouseevent
        self.last_artist = artist = event.artist
        #print('***picker handler:',artist, 'mouseevent:', mouseevent)
        if artist in self.arcs:
            #print('### selected arc',self.arcs[artist])
            self.picked = self.arcs[artist]
        elif artist in self.nodes:
            #print('### selected node',self.nodes[artist])
            self.picked = self.nodes[artist]
        elif artist==self.autoACtext:
            self.autoAC = True
            #print("*** autoAC")
        else:
            print("### unknown click")
```

4.1.4 Examples

In the following code `ne\_`, when given a number, returns a function that is true when its argument is not that number. For example, if `f=ne\_(3)`, then `f(2)` is True and `f(3)` is False. That is, `ne\_`$(x)(y)$ is true when $x \neq y$. Allowing a function of multiple arguments to use its arguments one at a time is called **currying**, after the logician Haskell Curry. Some alternative implementations are commented out; the uncommented one allows the partial functions to have names.

cspExamples.py — Example CSPs

```
from cspProblem import Variable, CSP, Constraint
from operator import lt,ne,eq,gt

def ne_(val):
    """not equal value"""
    # return lambda x: x != val # alternative definition
    # return partial(ne,val)  # another alternative definition
    def nev(x):
```

```
        return val != x
    nev.__name__ = f"{val} != "   # name of the function
    return nev
```

Similarly $is\_(x)(y)$ is true when $x = y$.

cspExamples.py — (continued)

```
def is_(val):
    """is a value"""
    # return lambda x: x == val # alternative definition
    # return partial(eq,val)   # another alternative definition
    def isv(x):
        return val == x
    isv.__name__ = f"{val} == "
    return isv
```

`csp0` has variables X, Y and Z, each with domain $\{1, 2, 3\}$. The constraints are $X < Y$ and $Y < Z$.

cspExamples.py — (continued)

```
X = Variable('X', {1,2,3}, position=(0.1,0.8))
Y = Variable('Y', {1,2,3}, position=(0.5,0.2))
Z = Variable('Z', {1,2,3}, position=(0.9,0.8))
csp0 = CSP("csp0", {X,Y,Z},
          [ Constraint([X,Y], lt, "X<Y"),
            Constraint([Y,Z], lt, "Y<Z")])
```

`csp1` has variables A, B and C, each with domain $\{1, 2, 3, 4\}$. The constraints are $A < B$, $B \neq 2$, and $B < C$. This is slightly more interesting than `csp0` as it has more solutions. This example is used in the unit tests, and so if it is changed, the unit tests need to be changed. `csp1s` is the same, but with only the constraints $A < B$ and $B < C$

cspExamples.py — (continued)

```
A = Variable('A', {1,2,3,4}, position=(0.2,0.9))
B = Variable('B', {1,2,3,4}, position=(0.8,0.9))
C = Variable('C', {1,2,3,4}, position=(1,0.3))
C0 = Constraint([A,B], lt, "A < B", position=(0.4,0.3))
C1 = Constraint([B], ne_(2), "B != 2", position=(1,0.7))
C2 = Constraint([B,C], lt, "B < C", position=(0.6,0.1))
csp1 = CSP("csp1", {A, B, C},
          [C0, C1, C2])

csp1s = CSP("csp1s", {A, B, C},
          [C0, C2]) # A<B, B<C
```

The next CSP, *csp*2 is Example 4.9 of Poole and Mackworth [2023]; the domain consistent network (after applying the unary constraints) is shown in Figure 4.2. Note that we use the same variables as the previous example and add two more.

cspExamples.py — (continued)

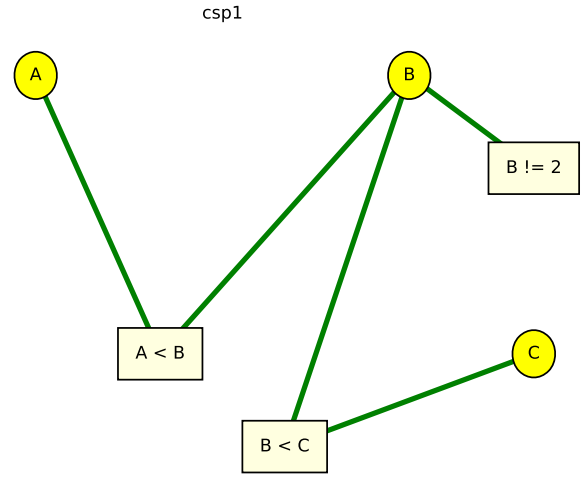


Figure 4.1: csp1.show()

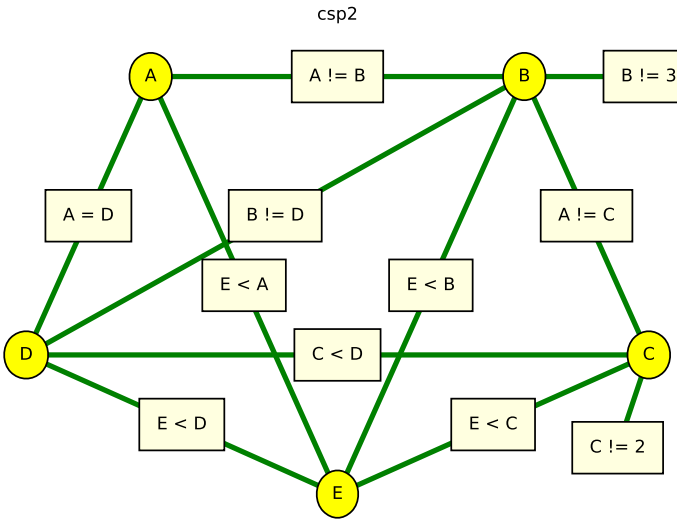


Figure 4.2: csp2.show()

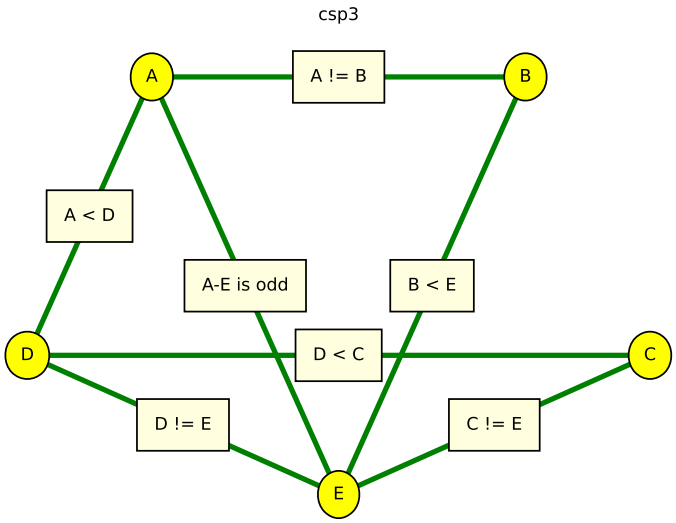


Figure 4.3: csp3.show()

```
D = Variable('D', {1,2,3,4}, position=(0,0.3))
E = Variable('E', {1,2,3,4}, position=(0.5,0))
csp2 = CSP("csp2", {A,B,C,D,E},
          [ Constraint([B], ne_(3), "B != 3", position=(1,0.9)),
           Constraint([C], ne_(2), "C != 2", position=(0.95,0.1)),
           Constraint([A,B], ne, "A != B"),
           Constraint([B,C], ne, "A != C"),
           Constraint([C,D], lt, "C < D"),
           Constraint([A,D], eq, "A = D"),
           Constraint([E,A], lt, "E < A"),
           Constraint([E,B], lt, "E < B"),
           Constraint([E,C], lt, "E < C"),
           Constraint([E,D], lt, "E < D"),
           Constraint([B,D], ne, "B != D")])
```

The following example is another scheduling problem (but with multiple answers). This is the same as "scheduling 2" in the original AIspace.org consistency app.

cspExamples.py — (continued)

```
csp3 = CSP("csp3", {A,B,C,D,E},
          [Constraint([A,B], ne, "A != B"),
           Constraint([A,D], lt, "A < D"),
           Constraint([A,E], lambda a,e: (a-e)%2 == 1, "A-E is odd"),
           Constraint([B,E], lt, "B < E"),
           Constraint([D,C], lt, "D < C"),
           Constraint([C,E], ne, "C != E"),
```

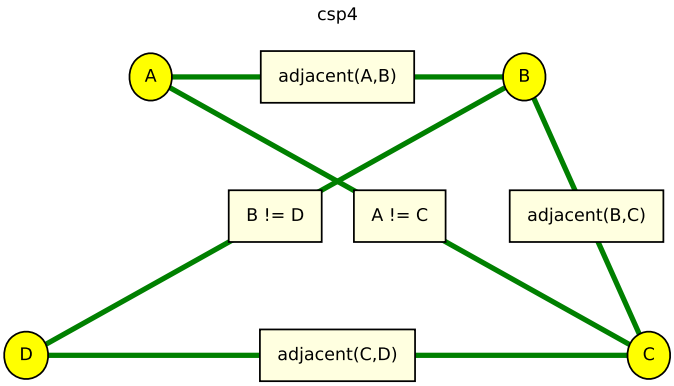


Figure 4.4: csp4.show()

```
73            Constraint([D,E], ne, "D != E")])
```

The following example is another abstract scheduling problem. What are the solutions?

cspExamples.py — (continued)

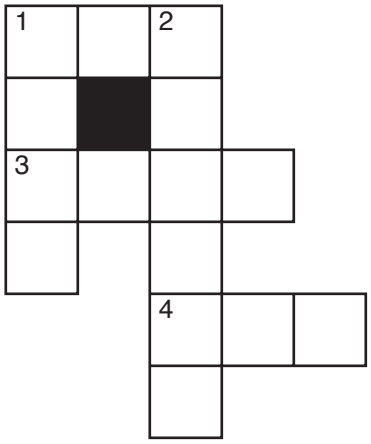
```
75 def adjacent(x,y):
76    """True when x and y are adjacent numbers"""
77    return abs(x-y) == 1
78
79 csp4 = CSP("csp4", {A,B,C,D},
80           [Constraint([A,B], adjacent, "adjacent(A,B)"),
81            Constraint([B,C], adjacent, "adjacent(B,C)"),
82            Constraint([C,D], adjacent, "adjacent(C,D)"),
83            Constraint([A,C], ne, "A != C"),
84            Constraint([B,D], ne, "B != D") ])
```

The following examples represent the crossword shown in Figure 4.5.

In the first representation, the variables represent words. The constraint imposed by the crossword is that where two words intersect, the letter at the intersection must be the same. The method `meet_at` is used to test whether two words intersect with the same letter. For example, the constraint `meet_at(2,0)` means that the third letter (at position 2) of the first argument is the same as the first letter of the second argument. This is shown in Figure 4.6.

cspExamples.py — (continued)

```
86 def meet_at(p1,p2):
```



Words:
ant, big, bus, car, has, book, buys, hold, lane, year, ginger, search, symbol, syntax.

Figure 4.5: crossword1: a crossword puzzle to be solved

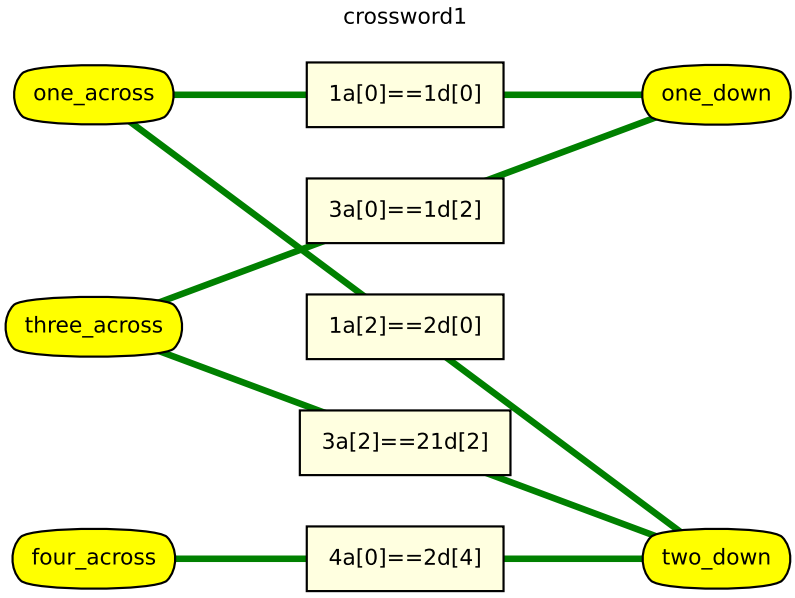


Figure 4.6: crossword1.show()

```
87      """returns a function of two words that is true
88                  when the words intersect at positions p1, p2.
89      The positions are relative to the words; starting at position 0.
90      meet_at(p1,p2)(w1,w2) is true if the same letter is at position p1 of
            word w1
91           and at position p2 of word w2.
92      """
93      def meets(w1,w2):
94          return w1[p1] == w2[p2]
95      meets.__name__ = f"meet_at({p1},{p2})"
96      return meets
97
98  one_across = Variable('one_across', {'ant', 'big', 'bus', 'car', 'has'},
        position=(0.1,0.9))
99  one_down = Variable('one_down', {'book', 'buys', 'hold', 'lane', 'year'},
        position=(0.9,0.9))
100 two_down = Variable('two_down', {'ginger', 'search', 'symbol', 'syntax'},
        position=(0.9,0.1))
101 three_across = Variable('three_across', {'book', 'buys', 'hold', 'land',
        'year'}, position=(0.1,0.5))
102 four_across = Variable('four_across',{'ant', 'big', 'bus', 'car', 'has'},
        position=(0.1,0.1))
103 crossword1 = CSP("crossword1",
104                 {one_across, one_down, two_down, three_across,
                        four_across},
105                 [Constraint([one_across,one_down],
                        meet_at(0,0),"1a[0]==1d[0]"),
106                  Constraint([one_across,two_down],
                        meet_at(2,0),"1a[2]==2d[0]"),
107                  Constraint([three_across,two_down],
                        meet_at(2,2),"3a[2]==21d[2]"),
108                  Constraint([three_across,one_down],
                        meet_at(0,2),"3a[0]==1d[2]"),
109                  Constraint([four_across,two_down],
                        meet_at(0,4),"4a[0]==2d[4]")
110                 ])
```

In an alternative representation of a crossword (the "dual" representation), the variables represent letters, and the constraints are that adjacent sequences of letters form words. This is shown in Figure 4.7.

cspExamples.py — (continued)

```
112 words = {'ant', 'big', 'bus', 'car', 'has','book', 'buys', 'hold',
113         'lane', 'year', 'ginger', 'search', 'symbol', 'syntax'}
114
115 def is_word(*letters, words=words):
116     """is true if the letters concatenated form a word in words"""
117     return "".join(letters) in words
118
119 letters = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
120   "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y",
```

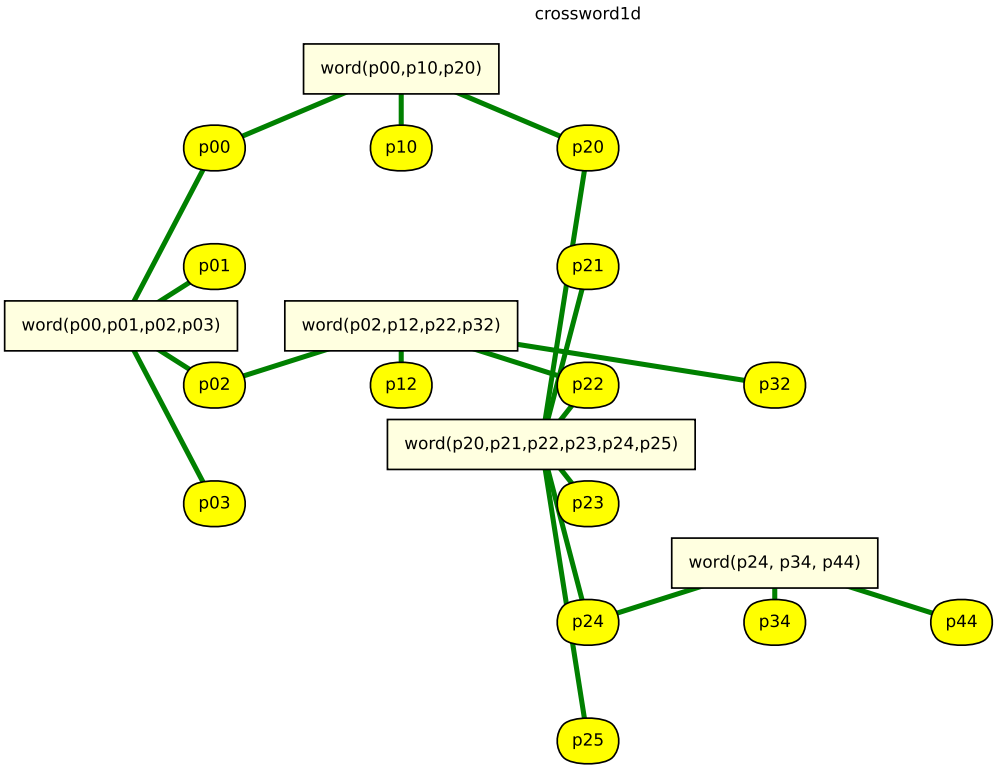


Figure 4.7: crossword1d.show()

```
121   "z"}
122
123 # pij is the variable representing the letter i from the left and j down
        (starting from 0)
124 p00 = Variable('p00', letters, position=(0.1,0.85))
125 p10 = Variable('p10', letters, position=(0.3,0.85))
126 p20 = Variable('p20', letters, position=(0.5,0.85))
127 p01 = Variable('p01', letters, position=(0.1,0.7))
128 p21 = Variable('p21', letters, position=(0.5,0.7))
129 p02 = Variable('p02', letters, position=(0.1,0.55))
130 p12 = Variable('p12', letters, position=(0.3,0.55))
131 p22 = Variable('p22', letters, position=(0.5,0.55))
132 p32 = Variable('p32', letters, position=(0.7,0.55))
133 p03 = Variable('p03', letters, position=(0.1,0.4))
134 p23 = Variable('p23', letters, position=(0.5,0.4))
135 p24 = Variable('p24', letters, position=(0.5,0.25))
136 p34 = Variable('p34', letters, position=(0.7,0.25))
137 p44 = Variable('p44', letters, position=(0.9,0.25))
138 p25 = Variable('p25', letters, position=(0.5,0.1))
139
140 crossword1d = CSP("crossword1d",
```

```
                {p00, p10, p20, # first row
                 p01, p21, # second row
                 p02, p12, p22, p32, # third row
                 p03, p23, #fourth row
                 p24, p34, p44, # fifth row
                 p25 # sixth row
                 },
                [Constraint([p00, p10, p20], is_word, "word(p00,p10,p20)",
                            position=(0.3,0.95)), #1-across
                 Constraint([p00, p01, p02, p03], is_word,
                     "word(p00,p01,p02,p03)",
                            position=(0,0.625)), # 1-down
                 Constraint([p02, p12, p22, p32], is_word,
                     "word(p02,p12,p22,p32)",
                            position=(0.3,0.625)), # 3-across
                 Constraint([p20, p21, p22, p23, p24, p25], is_word,
                     "word(p20,p21,p22,p23,p24,p25)",
                            position=(0.45,0.475)), # 2-down
                 Constraint([p24, p34, p44], is_word, "word(p24, p34,
                     p44)",
                            position=(0.7,0.325)) # 4-across
                 ])
```

Exercise 4.1 How many assignments of a value to each variable are there for each of the representations of the above crossword? Do you think an exhaustive enumeration will work for either one?

The queens problem is a puzzle on a chess board, where the idea is to place a queen on each column so the queens cannot take each other: there are no two queens on the same row, column or diagonal. The **n-queens problem** is a generalization where the size of the board is an $n \times n$, and n queens have to be placed.

Here is a representation of the n-queens problem, where the variables are the columns and the values are the rows in which the queen is placed. The original queens problem on a standard (8×8) chess board is `n_queens(8)`

cspExamples.py — (continued)

```
def queens(ri,rj):
    """ri and rj are different rows, return the condition that the queens
        cannot take each other"""
    def no_take(ci,cj):
        """is true if queen at (ri,ci) cannot take a queen at (rj,cj)"""
        return ci != cj and abs(ri-ci) != abs(rj-cj)
    return no_take

def n_queens(n):
    """returns a CSP for n-queens"""
    columns = list(range(n))
    variables = [Variable(f"R{i}",columns) for i in range(n)]
      # note positions will be random
```

```
172     return CSP("n-queens",
173                variables,
174                [Constraint([variables[i], variables[j]], queens(i,j),"" )
175                     for i in range(n) for j in range(n) if i != j])
176
177 # try the CSP n_queens(8) in one of the solvers.
178 # What is the smallest n for which there is a solution?
```

Exercise 4.2 How many constraints does this representation of the n-queens problem produce? Can it be done with fewer constraints? Either explain why it can't be done with fewer constraints, or give a solution using fewer constraints.

Unit tests

The following defines a **unit test** for csp solvers, by default using example csp1.

cspExamples.py — (continued)

```
180 def test_csp(CSP_solver, csp=csp1,
181             solutions=[{A: 1, B: 3, C: 4}, {A: 2, B: 3, C: 4}]):
182     """CSP_solver is a solver that takes a csp and returns a solution
183     csp is a constraint satisfaction problem
184     solutions is the list of all solutions to csp
185     This tests whether the solution returned by CSP_solver is a solution.
186     """
187     print("Testing csp with",CSP_solver.__doc__)
188     sol0 = CSP_solver(csp)
189     print("Solution found:",sol0)
190     assert sol0 in solutions, f"Solution not correct for {csp}"
191     print("Passed unit test")
```

Exercise 4.3 Modify *test* so that instead of taking in a list of solutions, it checks whether the returned solution actually is a solution.

Exercise 4.4 Propose a test that is appropriate for CSPs with no solutions. Assume that the test designer knows there are no solutions. Consider what a CSP solver should return if there are no solutions to the CSP.

Exercise 4.5 Write a unit test that checks whether all solutions (e.g., for the search algorithms that can return multiple solutions) are correct, and whether all solutions can be found.

4.2 A Simple Depth-first Solver

The first solver carries out a depth-first search through the space of partial assignments. This takes in a CSP problem and an optional variable ordering (a list of the variables in the CSP). It returns a generator of the solutions (see Section 1.5.3 on `yield` for enumerations).

cspDFS.py — Solving a CSP using depth-first search.

```
11 import cspExamples
```

```
12
13 def dfs_solver(constraints, context, var_order):
14     """generator for all solutions to csp.
15     context is an assignment of values to some of the variables.
16     var_order is  a list of the variables in csp that are not in context.
17     """
18     to_eval = {c for c in constraints if c.can_evaluate(context)}
19     if all(c.holds(context) for c in to_eval):
20         if var_order == []:
21             yield context
22         else:
23             rem_cons = [c for c in constraints if c not in to_eval]
24             var = var_order[0]
25             for val in var.domain:
26                 yield from dfs_solver(rem_cons, context|{var:val},
                       var_order[1:])
27
28 def dfs_solve_all(csp, var_order=None):
29     """depth-first CSP solver to return a list of all solutions to csp.
30     """
31     if var_order == None:  # use an arbitrary variable order
32         var_order = list(csp.variables)
33     return list( dfs_solver(csp.constraints, {}, var_order))
34
35 def dfs_solve1(csp, var_order=None):
36     """depth-first CSP solver"""
37     if var_order == None:  # use an arbitrary variable order
38         var_order = list(csp.variables)
39     for sol in dfs_solver(csp.constraints, {}, var_order):
40         return sol #return first one
41
42 if __name__ == "__main__":
43     cspExamples.test_csp(dfs_solve1)
44
45 #Try:
46 # dfs_solve_all(cspExamples.csp1)
47 # dfs_solve_all(cspExamples.csp2)
48 # dfs_solve_all(cspExamples.crossword1)
49 # dfs_solve_all(cspExamples.crossword1d) # warning: may take a *very* long
      time!
```

Exercise 4.6 Instead of testing all constraints at every node, change it so each constraint is only tested when all of its variables are assigned. Given an elimination ordering, it is possible to determine when each constraint needs to be tested. Implement this. Hint: create a parallel list of sets of constraints, where at each position i in the list, the constraints at position i can be evaluated when the variable at position i has been assigned.

Exercise 4.7 Estimate how long `dfs_solve_all(crossword1d)` will take on your computer. To do this, reduce the number of variables that need to be assigned, so that the simplified problem can be solved in a reasonable time (between 0.1

second and 10 seconds). This can be done by reducing the number of variables in `var_order`, as the program only splits on these. How much more time will it take if the number of variables is increased by 1? (Try it!) Then extrapolate to all of the variables. See Section 1.6.1 for how to time your code. Would making the code 100 times faster or using a computer 100 times faster help?

4.3 Converting CSPs to Search Problems

To run the demo, in folder "aipython", load "cspSearch.py", and copy and paste the example queries at the bottom of that file.

The next solver constructs a search space that can be solved using the search methods of the previous chapter. This takes in a CSP problem and an optional variable ordering, which is a list of the variables in the CSP. In this search space:

- A node is a *variable* : *value* dictionary which does not violate any constraints (so that dictionaries that violate any conmtratints are not added).
- An arc corresponds to an assignment of a value to the next variable. This assumes a static ordering; the next variable chosen to split does not depend on the context. If no variable ordering is given, this makes no attempt to choose a good ordering.

cspSearch.py — Representations of a Search Problem from a CSP.

```
11 from cspProblem import CSP, Constraint
12 from searchProblem import Arc, Search_problem
13
14 class Search_from_CSP(Search_problem):
15     """A search problem directly from the CSP.
16
17     A node is a variable:value dictionary"""
18     def __init__(self, csp, variable_order=None):
19         self.csp=csp
20         if variable_order:
21             assert set(variable_order) == set(csp.variables)
22             assert len(variable_order) == len(csp.variables)
23             self.variables = variable_order
24         else:
25             self.variables = list(csp.variables)
26
27     def is_goal(self, node):
28         """returns whether the current node is a goal for the search
29         """
30         return len(node)==len(self.csp.variables)
31
32     def start_node(self):
33         """returns the start node for the search
```

```
        """
        return {}
```

The *neighbors*(*node*) method uses the fact that the length of the node, which is the number of variables already assigned, is the index of the next variable to split on. Note that we do not need to check whether there are no more variables to split on, as the nodes are all consistent, by construction, and so when there are no more variables we have a solution, and so don't need the neighbors.

cspSearch.py — (continued)

```
    def neighbors(self, node):
        """returns a list of the neighboring nodes of node.
        """
        var = self.variables[len(node)] # the next variable
        res = []
        for val in var.domain:
            new_env = node|{var:val} #dictionary union
            if self.csp.consistent(new_env):
                res.append(Arc(node,new_env))
        return res
```

The unit tests relies on a solver. The following procedure creates a solver using search that can be tested.

cspSearch.py — (continued)

```
import cspExamples
from searchGeneric import Searcher

def solver_from_searcher(csp):
    """depth-first search solver"""
    path = Searcher(Search_from_CSP(csp)).search()
    if path is not None:
        return path.end()
    else:
        return None

if __name__ == "__main__":
    test_csp(solver_from_searcher)

## Test Solving CSPs with Search:
searcher1 = Searcher(Search_from_CSP(cspExamples.csp1))
#print(searcher1.search()) # get next solution
searcher2 = Searcher(Search_from_CSP(cspExamples.csp2))
#print(searcher2.search()) # get next solution
searcher3 = Searcher(Search_from_CSP(cspExamples.crossword1))
#print(searcher3.search()) # get next solution
searcher4 = Searcher(Search_from_CSP(cspExamples.crossword1d))
#print(searcher4.search()) # get next solution (warning: slow)
```

Exercise 4.8 What would happen if we constructed the new assignment by assigning $node[var] = val$ (with side effects) instead of using dictionary union? Give

an example of where this could give a wrong answer. How could the algorithm be changed to work with side effects? (Hint: think about what information needs to be in a node).

Exercise 4.9 Change neighbors so that it returns an iterator of values rather than a list. (Hint: use *yield*.)

4.4 Consistency Algorithms

To run the demo, in folder "aipython", load "cspConsistency.py", and copy and paste the commented-out example queries at the bottom of that file.

A *Con_solver* is used to simplify a CSP using arc consistency.

cspConsistency.py — Arc Consistency and Domain splitting for solving a CSP

```
from display import Displayable

class Con_solver(Displayable):
    """Solves a CSP with arc consistency and domain splitting
    """
    def __init__(self, csp):
        """a CSP solver that uses arc consistency
        * csp is the CSP to be solved
        """
        self.csp = csp
```

The following implementation of arc consistency maintains the set *to_do* of (variable, constraint) pairs that are to be checked. It takes in a domain dictionary and returns a new domain dictionary. It needs to be careful to avoid side effects; this is implemented here by copying the *domains* dictionary and the *to_do* set.

cspConsistency.py — (continued)

```
    def make_arc_consistent(self, domains=None, to_do=None):
        """Makes this CSP arc-consistent using generalized arc consistency
        domains is a variable:domain dictionary
        to_do is a set of (variable,constraint) pairs
        returns the reduced domains (an arc-consistent variable:domain
            dictionary)
        """
        if domains is None:
            self.domains = {var:var.domain for var in self.csp.variables}
        else:
            self.domains = domains.copy() # use a copy of domains
        if to_do is None:
            to_do = {(var, const) for const in self.csp.constraints
                    for var in const.scope}
        else:
```

```
36              to_do = to_do.copy() # use a copy of to_do
37          self.display(5,"Performing AC with domains", self.domains)
38          while to_do:
39              self.arc_selected = (var, const) = self.select_arc(to_do)
40              self.display(5, "Processing arc (", var, ",", const, ")")
41              other_vars = [ov for ov in const.scope if ov != var]
42              new_domain = {val for val in self.domains[var]
43                            if self.any_holds(self.domains, const, {var:
                                  val}, other_vars)}
44              if new_domain != self.domains[var]:
45                  self.add_to_do = self.new_to_do(var, const) - to_do
46                  self.display(3, f"Arc: ({var}, {const}) is inconsistent\n"
47                               f"Domain pruned, dom({var}) ={new_domain} due to
                                     {const}")
48                  self.domains[var] = new_domain
49                  self.display(4, " adding", self.add_to_do if self.add_to_do
50                                 else "nothing", "to to_do.")
51                  to_do |= self.add_to_do   # set union
52              self.display(5, f"Arc: ({var},{const}) now consistent")
53          self.display(5, "AC done. Reduced domains", self.domains)
54          return self.domains
55
56      def new_to_do(self, var, const):
57          """returns new elements to be added to to_do after assigning
58          variable var in constraint const.
59          """
60          return {(nvar, nconst) for nconst in self.csp.var_to_const[var]
61                  if nconst != const
62                  for nvar in nconst.scope
63                  if nvar != var}
```

The following selects an arc. Any element of *to_do* can be selected. The selected element needs to be removed from *to_do*. The default implementation just selects which ever element *pop* method for sets returns. The graphical user interface below allows the user to select an arc. Alternatively, a more sophisticated selection could be employed.

cspConsistency.py — (continued)

```
65      def select_arc(self, to_do):
66          """Selects the arc to be taken from to_do .
67          * to_do is a set of arcs, where an arc is a (variable,constraint)
                pair
68          the element selected must be removed from to_do.
69          """
70          return to_do.pop()
```

The value of `new_domain` is the subset of the domain of `var` that is consistent with the assignment to the other variables. To make it easier to understand, the following treats unary (with no other variables in the constraint) and binary (with one other variables in the constraint) constraints as special cases. These cases are not strictly necessary; the last case covers the first two cases, but is

more difficult to understand without seeing the first two cases. Note that this case analysis is not in the code distribution, but can replace the assignment to `new_domain` above.

```
            if len(other_vars)==0:          # unary constraint
                new_domain = {val for val in self.domains[var]
                              if const.holds({var:val})}
            elif len(other_vars)==1:        # binary constraint
                other = other_vars[0]
                new_domain = {val for val in self.domains[var]
                       if any(const.holds({var: val,other:other_val})
                              for other_val in self.domains[other])}
            else:                           # general case
                new_domain = {val for val in self.domains[var]
                    if self.any_holds(self.domains, const, {var: val}, other_vars)}
```

any_holds is a recursive function that tries to finds an assignment of values to the other variables (*other_vars*) that satisfies constraint *const* given the assignment in *env*. The integer variable *ind* specifies which index to *other_vars* needs to be checked next. As soon as one assignment returns *True,* the algorithm returns *True*.

cspConsistency.py — (continued)

```
72      def any_holds(self, domains, const, env, other_vars, ind=0):
73          """returns True if Constraint const holds for an assignment
74          that extends env with the variables in other_vars[ind:]
75          env is a dictionary
76          """
77          if ind == len(other_vars):
78              return const.holds(env)
79          else:
80              var = other_vars[ind]
81              for val in domains[var]:
82                  if self.any_holds(domains, const, env|{var:val}, other_vars,
                        ind + 1):
83                      return True
84              return False
```

4.4.1 Direct Implementation of Domain Splitting

The following is a direct implementation of domain splitting with arc consistency. It implements the generator interface of Python (see Section 1.5.3). When it has found a solution it `yield`s the result; otherwise it recursively splits a domain (using `yield from`).

cspConsistency.py — (continued)

```
86      def generate_sols(self, domains=None, to_do=None, context=dict()):
87          """return list of all solution to the current CSP
```

```
        to_do is the list of arcs to check
        context is a dictionary of splits made (used for display)
        """
        new_domains = self.make_arc_consistent(domains, to_do)
        if any(len(new_domains[var]) == 0 for var in new_domains):
            self.display(1,f"No solutions for context {context}")
        elif all(len(new_domains[var]) == 1 for var in new_domains):
            self.display(1, "solution:", str({var: select(
                new_domains[var]) for var in new_domains}))
            yield {var: select(new_domains[var]) for var in new_domains}
        else:
            var = self.select_var(x for x in self.csp.variables if
                len(new_domains[x]) > 1)
            dom1, dom2 = partition_domain(new_domains[var])
            self.display(5, "...splitting", var, "into", dom1, "and", dom2)
            new_doms1 = new_domains | {var:dom1}
            new_doms2 = new_domains | {var:dom2}
            to_do = self.new_to_do(var, None)
            self.display(4, " adding", to_do if to_do else "nothing", "to
                to_do.")
            yield from self.generate_sols(new_doms1, to_do,
                context|{var:dom1})
            yield from self.generate_sols(new_doms2, to_do,
                context|{var:dom1})

    def solve_all(self, domains=None, to_do=None):
        return list(self.generate_sols())

    def solve_one(self, domains=None, to_do=None):
        return select(self.generate_sols())

    def select_var(self, iter_vars):
        """return the next variable to split"""
        return select(iter_vars)

def partition_domain(dom):
    """partitions domain dom into two.
    """
    split = len(dom) // 2
    dom1 = set(list(dom)[:split])
    dom2 = dom - dom1
    return dom1, dom2
```

cspConsistency.py — (continued)

```
def select(iterable):
    """select an element of iterable.
    Returns None if there is no such element.

    This implementation just picks the first element.
    For many uses, which element is selected does not affect correctness,
```

```
133     but may affect efficiency.
134     """
135     for e in iterable:
136         return e # returns first element found
```

Exercise 4.10 Implement *solve_all* that returns the set of all solutions without using `yield`. Hint: it can be like `generate_sols` but returns a set of solutions; the recursive calls can be unioned; `|` is Python's union.

Exercise 4.11 Implement *solve_one* that returns one solution if one exists, or `False` otherwise, without using `yield`. Hint: Python's "or" has the behavior `A or B` will return the value of `A` unless it is None or False, in which case the value of `B` is returned.

Unit test:

cspConsistency.py — (continued)

```
138 import cspExamples
139 def ac_solver(csp):
140     "arc consistency (ac_solver)"
141     for sol in Con_solver(csp).generate_sols():
142         return sol
143 
144 if __name__ == "__main__":
145     cspExamples.test_csp(ac_solver)
```

4.4.2 Consistency GUI

The consistency GUI allows students to step through the algorithm, choosing which arc to process next, and which variable to split.

Figure 4.8 shows the state of the GUI after two arcs have been made arc consistent. The arcs on the `to_do` list arc colored blue. The green arcs are those have been made arc consistent. The user can click on a blue arc to process that arc. If the arc selected is not arc consistent, it is made red, the domain is reduced, and then the arc becomes green. If the arc was already arc consistent it turns green.

This is implemented by overriding `select_arc` and `select_var` to allow the user to pick the arcs and the variables, and overriding `display` to allow for the animation. Note that the first argument of `display` (the number) in the code above is interpreted with a special meaning by the GUI and should only be changed with care.

Clicking `AutoAC` automates arc selection until the network is arc consistent.

cspConsistencyGUI.py — GUI for consistency-based CSP solving

```
11 from cspConsistency import Con_solver
12 import matplotlib.pyplot as plt
13 
14 class ConsistencyGUI(Con_solver):
15     def __init__(self, csp, fontsize=10, speed=1, **kwargs):
```

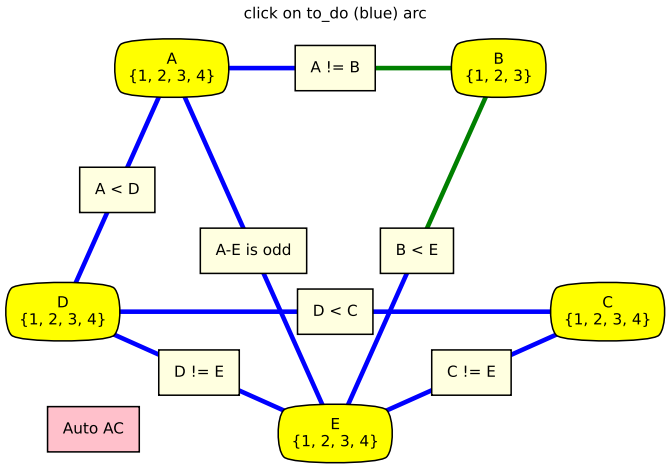


Figure 4.8: ConsistencyGUI(cspExamples.csp3).go()

```
        """
        csp is the csp to show
        fontsize is the size of the text
        speed is the number of animations per second (controls delay_time)
            1 (slow) and 4 (fast) seem like good values
        """
        self.fontsize = fontsize
        self.delay_time = 1/speed
        self.quitting = False
        Con_solver.__init__(self, csp, **kwargs)
        csp.show(showAutoAC = True)
        csp.fig.canvas.mpl_connect('close_event', self.window_closed)

    def go(self):
        try:
            res = self.solve_all()
            self.csp.draw_graph(domains=self.domains,
                                title="No more solutions. GUI finished. ",
                                fontsize=self.fontsize)
            return res
        except ExitToPython:
            print("GUI closed")

    def select_arc(self, to_do):
        while True:
            self.csp.draw_graph(domains=self.domains, to_do=to_do,
                                title="click on to_do (blue) arc",
                                    fontsize=self.fontsize)
            self.wait_for_user()
```

```
            if self.csp.autoAC:
                break
            picked = self.csp.picked
            self.csp.picked = None
            if picked in to_do:
                to_do.remove(picked)
                print(f"{picked} picked")
                return picked
            else:
                print(f"{picked} not in to_do. Pick one of {to_do}")
        if self.csp.autoAC:
            self.csp.draw_graph(domains=self.domains, to_do=to_do,
                                title="Auto AC", fontsize=self.fontsize)
            plt.pause(self.delay_time)
            return to_do.pop()

    def select_var(self, iter_vars):
        vars = list(iter_vars)
        while True:
            self.csp.draw_graph(domains=self.domains,
                                title="Arc consistent. Click node to
                                    split",
                                fontsize=self.fontsize)
            self.csp.autoAC = False
            self.wait_for_user()
            picked = self.csp.picked
            self.csp.picked = None
            if picked in vars:
                #print("splitting",picked)
                return picked
            else:
                print(picked,"not in",vars)

    def display(self,n,*args,**nargs):
        if n <= self.max_display_level: # default display
            print(*args, **nargs)
        if n==1: # solution found or no solutions"
            self.csp.draw_graph(domains=self.domains, to_do=set(),
                                title=' '.join(args)+": click any node or
                                    arc to continue",
                                fontsize=self.fontsize)
            self.csp.autoAC = False
            self.wait_for_user()
            self.csp.picked = None
        elif n==2: # backtracking
            plt.title("backtracking: click any node or arc to continue")
            self.csp.autoAC = False
            self.wait_for_user()
            self.csp.picked = None
        elif n==3: # inconsistent arc
```

```
92              line = self.csp.thelines[self.arc_selected]
93              line.set_color('red')
94              line.set_linewidth(10)
95              plt.pause(self.delay_time)
96              line.set_color('limegreen')
97              line.set_linewidth(self.csp.linewidth)
98          #elif n==4 and self.add_to_do: # adding to to_do
99          #    print("adding to to_do",self.add_to_do) ## highlight these arc
100
101     def wait_for_user(self):
102         while self.csp.picked == None and not self.csp.autoAC and not
                self.quitting:
103             plt.pause(0.01) # controls reaction time of GUI
104         if self.quitting:
105             raise ExitToPython()
106
107     def window_closed(self, event):
108         self.quitting = True
109
110 class ExitToPython(Exception):
111     pass
112
113 import cspExamples
114 # Try:
115 # ConsistencyGUI(cspExamples.csp1).go()
116 # ConsistencyGUI(cspExamples.csp3).go()
117 # ConsistencyGUI(cspExamples.csp3, speed=4, fontsize=15).go()
118
119 if __name__ == "__main__":
120     print("Try e.g.: ConsistencyGUI(cspExamples.csp3).go()")
```

4.4.3 Domain Splitting as an interface to graph searching

An alternative implementation is to implement domain splitting in terms of the search abstraction of Chapter 3.

A node is a dictionary that maps the variables to their (pruned) domains..

cspConsistency.py — (continued)

```
147 from searchProblem import Arc, Search_problem
148
149 class Search_with_AC_from_CSP(Search_problem,Displayable):
150     """A search problem with arc consistency and domain splitting
151
152     A node is a CSP """
153     def __init__(self, csp):
154         self.cons = Con_solver(csp) #copy of the CSP
155         self.domains = self.cons.make_arc_consistent()
156
157     def is_goal(self, node):
```

```
158         """node is a goal if all domains have 1 element"""
159         return all(len(node[var])==1 for var in node)
160 
161     def start_node(self):
162         return self.domains
163 
164     def neighbors(self,node):
165         """returns the neighboring nodes of node.
166         """
167         neighs = []
168         var = select(x for x in node if len(node[x])>1)
169         if var:
170             dom1, dom2 = partition_domain(node[var])
171             self.display(2,"Splitting", var, "into", dom1, "and", dom2)
172             to_do = self.cons.new_to_do(var,None)
173             for dom in [dom1,dom2]:
174                 newdoms = node | {var:dom}
175                 cons_doms = self.cons.make_arc_consistent(newdoms,to_do)
176                 if all(len(cons_doms[v])>0 for v in cons_doms):
177                     # all domains are non-empty
178                     neighs.append(Arc(node,cons_doms))
179                 else:
180                     self.display(2,"...",var,"in",dom,"has no solution")
181         return neighs
```

Exercise 4.12 When splitting a domain, this code splits the domain into half, approximately in half (without any effort to make a sensible choice). Does it work better to split one element from a domain?

Unit test:

cspConsistency.py — (continued)

```
183 import cspExamples
184 from searchGeneric import Searcher
185 
186 def ac_search_solver(csp):
187     """arc consistency (search interface)"""
188     sol = Searcher(Search_with_AC_from_CSP(csp)).search()
189     if sol:
190         return {v:select(d) for (v,d) in sol.end().items()}
191 
192 if __name__ == "__main__":
193     cspExamples.test_csp(ac_search_solver)
```

Testing:

cspConsistency.py — (continued)

```
195 ## Test Solving CSPs with Arc consistency and domain splitting:
196 #Con_solver.max_display_level = 4 # display details of AC (0 turns off)
197 #Con_solver(cspExamples.csp1).solve_all()
198 #searcher1d = Searcher(Search_with_AC_from_CSP(cspExamples.csp1))
```

```
#print(searcher1d.search())
#Searcher.max_display_level = 2 # display search trace (0 turns off)
#searcher2c = Searcher(Search_with_AC_from_CSP(cspExamples.csp2))
#print(searcher2c.search())
#searcher3c = Searcher(Search_with_AC_from_CSP(cspExamples.crossword1))
#print(searcher3c.search())
#searcher4c = Searcher(Search_with_AC_from_CSP(cspExamples.crossword1d))
#print(searcher4c.search())
```

4.5 Solving CSPs using Stochastic Local Search

To run the demo, in folder "aipython", load "cspSLS.py", and copy and paste the commented-out example queries at the bottom of that file. This assumes Python 3. Some of the queries require matplotlib.

The following code implements the two-stage choice (select one of the variables that are involved in the most constraints that are violated, then a value), the any-conflict algorithm (select a variable that participates in a violated constraint) and a random choice of variable, as well as a probabilistic mix of the three.

Given a CSP, the stochastic local searcher (*SLSearcher*) creates the data structures:

- *variables_to_select* is the set of all of the variables with domain-size greater than one. For a variable not in this set, we cannot pick another value from that variable.
- *var_to_constraints* maps from a variable into the set of constraints it is involved in. Note that the inverse mapping from constraints into variables is part of the definition of a constraint.

cspSLS.py — Stochastic Local Search for Solving CSPs

```
from cspProblem import CSP, Constraint
from searchProblem import Arc, Search_problem
from display import Displayable
import random
import heapq

class SLSearcher(Displayable):
    """A search problem directly from the CSP..

    A node is a variable:value dictionary"""
    def __init__(self, csp):
        self.csp = csp
        self.variables_to_select = {var for var in self.csp.variables
                                    if len(var.domain) > 1}
```

```
25        # Create assignment and conflicts set
26        self.current_assignment = None # this will trigger a random restart
27        self.number_of_steps = 0 #number of steps after the initialization
```

restart creates a new total assignment, and constructs the set of conflicts (the constraints that are false in this assignment).

cspSLS.py — (continued)

```
29    def restart(self):
30        """creates a new total assignment and the conflict set
31        """
32        self.current_assignment = {var:random_choice(var.domain) for
33                                   var in self.csp.variables}
34        self.display(2,"Initial assignment",self.current_assignment)
35        self.conflicts = set()
36        for con in self.csp.constraints:
37            if not con.holds(self.current_assignment):
38                self.conflicts.add(con)
39        self.display(2,"Number of conflicts",len(self.conflicts))
40        self.variable_pq = None
```

The *search* method is the top-level searching algorithm. It can either be used to start the search or to continue searching. If there is no current assignment, it must create one. Note that, when counting steps, a restart is counted as one step, which is not appropriate for CSPs with many variables, as it is a relatively expensive operation for these cases.

This method selects one of two implementations. The argument *prob_best* is the probability of selecting a best variable (one involving the most conflicts). When the value of *prob_best* is positive, the algorithm needs to maintain a priority queue of variables and the number of conflicts (using *search_with_var_pq*). If the probability of selecting a best variable is zero, it does not need to maintain this priority queue (as implemented in *search_with_any_conflict*).

The argument *prob_anycon* is the probability that the any-conflict strategy is used (which selects a variable at random that is in a conflict), assuming that it is not picking a best variable. Note that for the probability parameters, any value less that zero acts like probability zero and any value greater than 1 acts like probability 1. This means that when $prob\_anycon = 1.0$, a best variable is chosen with probability *prob_best*, otherwise a variable in any conflict is chosen. A variable is chosen at random with probability $1 - prob\_anycon - prob\_best$ as long as that is positive.

This returns the number of steps needed to find a solution, or *None* if no solution is found. If there is a solution, it is in *self.current_assignment*.

cspSLS.py — (continued)

```
42    def search(self,max_steps, prob_best=0, prob_anycon=1.0):
43        """
44        returns the number of steps or None if these is no solution.
45        If there is a solution, it can be found in self.current_assignment
46
```

```
47          max_steps is the maximum number of steps it will try before giving
                up
48          prob_best is the probability that a best variable (one in most
                conflict) is selected
49          prob_anycon is the probability that a variable in any conflict is
                selected
50          (otherwise a variable is chosen at random)
51          """
52          if self.current_assignment is None:
53              self.restart()
54              self.number_of_steps += 1
55              if not self.conflicts:
56                  self.display(1,"Solution found:", self.current_assignment,
                        "after restart")
57                  return self.number_of_steps
58          if prob_best > 0: # we need to maintain a variable priority queue
59              return self.search_with_var_pq(max_steps, prob_best,
                    prob_anycon)
60          else:
61              return self.search_with_any_conflict(max_steps, prob_anycon)
```

Exercise 4.13 This does an initial random assignment but does not do any random restarts. Implement a searcher that takes in the maximum number of walk steps (corresponding to existing *max_steps*) and the maximum number of restarts, and returns the total number of steps for the first solution found. (As in *search*, the solution found can be extracted from the variable *self.current_assignment*).

4.5.1 Any-conflict

In the any-conflict heuristic a variable that participates in a violated constraint is picked at random. The implementation need to keeps track of which variables are in conflicts. This is can avoid the need for a priority queue that is needed when the probability of picking a best variable is greater than zero.

cspSLS.py — (continued)

```
63      def search_with_any_conflict(self, max_steps, prob_anycon=1.0):
64          """Searches with the any_conflict heuristic.
65          This relies on just maintaining the set of conflicts;
66          it does not maintain a priority queue
67          """
68          self.variable_pq = None # we are not maintaining the priority queue.
69                                  # This ensures it is regenerated if
70                                  #   we call search_with_var_pq.
71          for i in range(max_steps):
72              self.number_of_steps +=1
73              if  random.random() < prob_anycon:
74                  con = random_choice(self.conflicts) # pick random conflict
75                  var = random_choice(con.scope) # pick variable in conflict
76              else:
77                  var = random_choice(self.variables_to_select)
```

```
78            if len(var.domain) > 1:
79                val = random_choice([val for val in var.domain
80                                     if val is not
                                         self.current_assignment[var]])
81                self.display(2,self.number_of_steps,":
                      Assigning",var,"=",val)
82                self.current_assignment[var]=val
83                for varcon in self.csp.var_to_const[var]:
84                    if varcon.holds(self.current_assignment):
85                        if varcon in self.conflicts:
86                            self.conflicts.remove(varcon)
87                    else:
88                        if varcon not in self.conflicts:
89                            self.conflicts.add(varcon)
90                self.display(2,"   Number of conflicts",len(self.conflicts))
91            if not self.conflicts:
92                self.display(1,"Solution found:", self.current_assignment,
93                               "in", self.number_of_steps,"steps")
94                return self.number_of_steps
95        self.display(1,"No solution in",self.number_of_steps,"steps",
96                    len(self.conflicts),"conflicts remain")
97        return None
```

Exercise 4.14 This makes no attempt to find the best value for the variable selected. Modify the code to include an option selects a value for the selected variable that reduces the number of conflicts the most. Have a parameter that specifies the probability that the best value is chosen, and otherwise chooses a value at random.

4.5.2 Two-Stage Choice

This is the top-level searching algorithm that maintains a priority queue of variables ordered by the number of conflicts, so that the variable with the most conflicts is selected first. If there is no current priority queue of variables, one is created.

The main complexity here is to maintain the priority queue. When a variable `var` is assigned a value `val`, for each constraint that has become satisfied or unsatisfied, each variable involved in the constraint need to have its count updated. The change is recorded in the dictionary *var_differential*, which is used to update the priority queue (see Section 4.5.3).

cspSLS.py — (continued)

```
99    def search_with_var_pq(self,max_steps, prob_best=1.0, prob_anycon=1.0):
100       """search with a priority queue of variables.
101       This is used to select a variable with the most conflicts.
102       """
103       if not self.variable_pq:
104           self.create_pq()
105       pick_best_or_con = prob_best + prob_anycon
```

```
        for i in range(max_steps):
            self.number_of_steps +=1
            randnum = random.random()
            ## Pick a variable
            if randnum < prob_best: # pick best variable
                var,oldval = self.variable_pq.top()
            elif randnum < pick_best_or_con: # pick a variable in a conflict
                con = random_choice(self.conflicts)
                var = random_choice(con.scope)
            else: #pick any variable that can be selected
                var = random_choice(self.variables_to_select)
            if len(var.domain) > 1: # var has other values
                ## Pick a value
                val = random_choice([val for val in var.domain if val is not
                                     self.current_assignment[var]])
                self.display(2,"Assigning",var,val)
                ## Update the priority queue
                var_differential = {}
                self.current_assignment[var]=val
                for varcon in self.csp.var_to_const[var]:
                    self.display(3,"Checking",varcon)
                    if varcon.holds(self.current_assignment):
                        if varcon in self.conflicts: # became consistent
                            self.display(3,"Became consistent",varcon)
                            self.conflicts.remove(varcon)
                            for v in varcon.scope: # v is in one fewer
                                conflicts
                                var_differential[v] =
                                    var_differential.get(v,0)-1
                    else:
                        if varcon not in self.conflicts: # was consis, not now
                            self.display(3,"Became inconsistent",varcon)
                            self.conflicts.add(varcon)
                            for v in varcon.scope: # v is in one more
                                conflicts
                                var_differential[v] =
                                    var_differential.get(v,0)+1
                self.variable_pq.update_each_priority(var_differential)
                self.display(2,"Number of conflicts",len(self.conflicts))
            if not self.conflicts: # no conflicts, so solution found
                self.display(1,"Solution found:",
                    self.current_assignment,"in",
                            self.number_of_steps,"steps")
                return self.number_of_steps
        self.display(1,"No solution in",self.number_of_steps,"steps",
                    len(self.conflicts),"conflicts remain")
        return None
```

create_pq creates an updatable priority queue of the variables, ordered by the number of conflicts they participate in. The priority queue only includes variables in conflicts and the value of a variable is the *negative* of the number of

conflicts the variable is in. This ensures that the priority queue, which picks the minimum value, picks a variable with the most conflicts.

cspSLS.py — (continued)

```
149     def create_pq(self):
150         """Create the variable to number-of-conflicts priority queue.
151         This is needed to select the variable in the most conflicts.
152 
153         The value of a variable in the priority queue is the negative of the
154         number of conflicts the variable appears in.
155         """
156         self.variable_pq = Updatable_priority_queue()
157         var_to_number_conflicts = {}
158         for con in self.conflicts:
159             for var in con.scope:
160                 var_to_number_conflicts[var] =
                        var_to_number_conflicts.get(var,0)+1
161         for var,num in var_to_number_conflicts.items():
162             if num>0:
163                 self.variable_pq.add(var,-num)
```

cspSLS.py — (continued)

```
165 def random_choice(st):
166     """selects a random element from set st.
167     It would be more efficient to convert to a tuple or list only once
168     (left as exercise)."""
169     return random.choice(tuple(st))
```

Exercise 4.15 These implementations always select a value for the variable selected that is different from its current value (if that is possible). Change the code so that it does not have this restriction (so it can leave the value the same). Would you expect this code to be faster? Does it work worse (or better)?

4.5.3 Updatable Priority Queues

An **updatable priority queue** is a priority queue, where key-value pairs can be stored, and the pair with the smallest key can be found and removed quickly, and where the values can be updated. This implementation follows the idea of `http://docs.python.org/3.9/library/heapq.html`, where the updated elements are marked as removed. This means that the priority queue can be used unmodified. However, this might be expensive if changes are more common than popping (as might happen if the probability of choosing the best is close to zero).

In this implementation, the equal values are sorted randomly. This is achieved by having the elements of the heap being $[val, rand, elt]$ triples, where the second element is a random number. Note that Python requires this to be a list, not a tuple, as the tuple cannot be modified.

cspSLS.py — (continued)

```
class Updatable_priority_queue(object):
    """A priority queue where the values can be updated.
    Elements with the same value are ordered randomly.

    This code is based on the ideas described in
    http://docs.python.org/3.3/library/heapq.html
    It could probably be done more efficiently by
    shuffling the modified element in the heap.
    """
    def __init__(self):
        self.pq = []  # priority queue of [val,rand,elt] triples
        self.elt_map = {} # map from elt to [val,rand,elt] triple in pq
        self.REMOVED = "*removed*" # a string that won't be a legal element
        self.max_size=0

    def add(self,elt,val):
        """adds elt to the priority queue with priority=val.
        """
        assert val <= 0,val
        assert elt not in self.elt_map, elt
        new_triple = [val, random.random(),elt]
        heapq.heappush(self.pq, new_triple)
        self.elt_map[elt] = new_triple

    def remove(self,elt):
        """remove the element from the priority queue"""
        if elt in self.elt_map:
            self.elt_map[elt][2] = self.REMOVED
            del self.elt_map[elt]

    def update_each_priority(self,update_dict):
        """update values in the priority queue by subtracting the values in
        update_dict from the priority of those elements in priority queue.
        """
        for elt,incr in update_dict.items():
            if incr != 0:
                newval = self.elt_map.get(elt,[0])[0] - incr
                assert newval <= 0, f"{elt}:{newval+incr}-{incr}"
                self.remove(elt)
                if newval != 0:
                    self.add(elt,newval)

    def pop(self):
        """Removes and returns the (elt,value) pair with minimal value.
        If the priority queue is empty, IndexError is raised.
        """
        self.max_size = max(self.max_size, len(self.pq)) # keep statistics
        triple = heapq.heappop(self.pq)
        while triple[2] == self.REMOVED:
```

```
            triple = heapq.heappop(self.pq)
        del self.elt_map[triple[2]]
        return triple[2], triple[0] # elt, value

    def top(self):
        """Returns the (elt,value) pair with minimal value, without
            removing it.
        If the priority queue is empty, IndexError is raised.
        """
        self.max_size = max(self.max_size, len(self.pq)) # keep statistics
        triple = self.pq[0]
        while triple[2] == self.REMOVED:
            heapq.heappop(self.pq)
            triple = self.pq[0]
        return triple[2], triple[0] # elt, value

    def empty(self):
        """returns True iff the priority queue is empty"""
        return all(triple[2] == self.REMOVED for triple in self.pq)
```

4.5.4 Plotting Run-Time Distributions

Runtime_distribution uses matplotlib to plot run time distributions. Here the run time is a misnomer as we are only plotting the number of steps, not the time. Computing the run time is non-trivial as many of the runs have a very short run time. To compute the time accurately would require running the same code, with the same random seed, multiple times to get a good estimate of the run time. This is left as an exercise.

cspSLS.py — (continued)

```
import matplotlib.pyplot as plt
# plt.style.use('grayscale')

class Runtime_distribution(object):
    def __init__(self, csp, xscale='log'):
        """Sets up plotting for csp
        xscale is either 'linear' or 'log'
        """
        self.csp = csp
        plt.ion()
        self.fig, self.ax = plt.subplots()
        self.ax.set_xlabel("Number of Steps")
        self.ax.set_ylabel("Cumulative Number of Runs")
        self.ax.set_xscale(xscale) # Makes a 'log' or 'linear' scale

    def plot_runs(self,num_runs=100,max_steps=1000, prob_best=1.0,
        prob_anycon=1.0):
        """Plots num_runs of SLS for the given settings.
        """
```

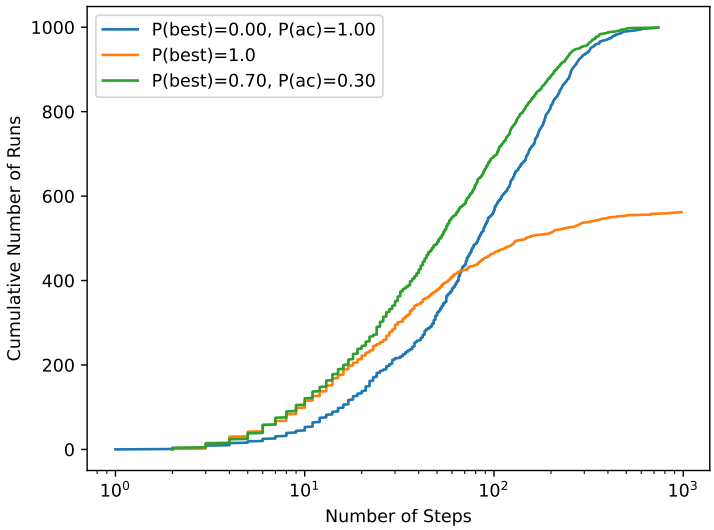


Figure 4.9: Run-time distributions for three algorithms on *csp*2.

```
257         stats = []
258         SLSearcher.max_display_level, temp_mdl = 0,
                SLSearcher.max_display_level # no display
259         for i in range(num_runs):
260             searcher = SLSearcher(self.csp)
261             num_steps = searcher.search(max_steps, prob_best, prob_anycon)
262             if num_steps:
263                 stats.append(num_steps)
264         stats.sort()
265         if prob_best >= 1.0:
266             label = "P(best)=1.0"
267         else:
268             p_ac = min(prob_anycon, 1-prob_best)
269             label = "P(best)=%.2f, P(ac)=%.2f" % (prob_best, p_ac)
270         self.ax.plot(stats,range(len(stats)),label=label)
271         self.ax.legend(loc="upper left")
272         SLSearcher.max_display_level= temp_mdl #restore display
```

Figure 4.9 gives run-time distributions for 3 algorithms. It is also useful to compare the distributions of different runs of the same algorithms and settings.

4.5.5 Testing

cspSLS.py — (continued)

```
274 import cspExamples
```

```
275 def sls_solver(csp,prob_best=0.7):
276     """stochastic local searcher (prob_best=0.7)"""
277     se0 = SLSearcher(csp)
278     se0.search(1000,prob_best)
279     return se0.current_assignment
280 def any_conflict_solver(csp):
281     """stochastic local searcher (any-conflict)"""
282     return sls_solver(csp,0)
283 
284 if __name__ == "__main__":
285     cspExamples.test_csp(sls_solver)
286     cspExamples.test_csp(any_conflict_solver)
287 
288 ## Test Solving CSPs with Search:
289 #se1 = SLSearcher(cspExamples.csp1); print(se1.search(100))
290 #se2 = SLSearcher(cspExamples.csp2); print(se2.search(1000,1.0)) # greedy
291 #se2 = SLSearcher(cspExamples.csp2); print(se2.search(1000,0)) #
        any_conflict
292 #se2 = SLSearcher(cspExamples.csp2); print(se2.search(1000,0.7)) # 70%
        greedy; 30% any_conflict
293 #SLSearcher.max_display_level=2 #more detailed display
294 #se3 = SLSearcher(cspExamples.crossword1); print(se3.search(100),0.7)
295 #p = Runtime_distribution(cspExamples.csp2)
296 #p.plot_runs(1000,1000,0) # any_conflict
297 #p.plot_runs(1000,1000,1.0) # greedy
298 #p.plot_runs(1000,1000,0.7) # 70% greedy; 30% any_conflict
```

Exercise 4.16 Modify this to plot the run time, instead of the number of steps. To measure run time use *timeit* (`https://docs.python.org/3.9/library/timeit.html`). Small run times are inaccurate, so timeit can run the same code multiple times. Stochastic local algorithms give different run times each time called. To make the timing meaningful, you need to make sure the random seed is the same for each repeated call (see `random.getstate` and `random.setstate` in `https://docs.python.org/3.9/library/random.html`). Because the run time for different seeds can vary a great deal, for each seed, you should start with 1 iteration and multiplying it by, say 10, until the time is greater than 0.2 seconds. Make sure you plot the average time for each run. Before you start, try to estimate the total run time, so you will be able to tell if there is a problem with the algorithm stopping.

4.6 Discrete Optimization

A `SoftConstraint` is a constraint, but where the condition is a real-valued cost function. The aim is to find the assignment with the lowest sum of costs. Because the definition of the constraint class did not force the condition to be Boolean, you can use the `Constraint` class for soft constraints too.

cspSoft.py — Representations of Soft Constraints

```
11 from cspProblem import Variable, Constraint, CSP
12 class SoftConstraint(Constraint):
```

```
13      """A Constraint consists of
14      * scope: a tuple of variables
15      * function: a real-valued costs function that can applied to a tuple of
            values
16      * string: a string for printing the constraints. All of the strings
            must be unique.
17      for the variables
18      """
19      def __init__(self, scope, function, string=None, position=None):
20          Constraint.__init__(self, scope, function, string, position)
21
22      def value(self,assignment):
23          return self.holds(assignment)
```

cspSoft.py — (continued)

```
25  A = Variable('A', {1,2}, position=(0.2,0.9))
26  B = Variable('B', {1,2,3}, position=(0.8,0.9))
27  C = Variable('C', {1,2}, position=(0.5,0.5))
28  D = Variable('D', {1,2}, position=(0.8,0.1))
29
30  def c1fun(a,b):
31      if a==1: return (5 if b==1 else 2)
32      else: return (0 if b==1 else 4 if b==2 else 3)
33  c1 = SoftConstraint([A,B],c1fun,"c1")
34  def c2fun(b,c):
35      if b==1: return (5 if c==1 else 2)
36      elif b==2: return (0 if c==1 else 4)
37      else: return (2 if c==1 else 0)
38  c2 = SoftConstraint([B,C],c2fun,"c2")
39  def c3fun(b,d):
40      if b==1: return (3 if d==1 else 0)
41      elif b==2: return 2
42      else: return (2 if d==1 else 4)
43  c3 = SoftConstraint([B,D],c3fun,"c3")
44
45  def penalty_if_same(pen):
46      "returns a function that gives a penalty of pen if the arguments are
            the same"
47      return lambda x,y: (pen if (x==y) else 0)
48
49  c4 = SoftConstraint([C,A],penalty_if_same(3),"c4")
50
51  scsp1 = CSP("scsp1", {A,B,C,D}, [c1,c2,c3,c4])
52
53  ### The second soft CSP has an extra variable, and 2 constraints
54  E = Variable('E', {1,2}, position=(0.1,0.1))
55
56  c5 = SoftConstraint([C,E],penalty_if_same(3),"c5")
57  c6 = SoftConstraint([D,E],penalty_if_same(2),"c6")
58  scsp2 = CSP("scsp1", {A,B,C,D,E}, [c1,c2,c3,c4,c5,c6])
```

4.6.1 Branch-and-bound Search

Here we specialize the branch-and-bound algorithm (Section 3.3 on page 65) to solve soft CSP problems.

cspSoft.py — (continued)

```
60 from display import Displayable
61 import math
62
63 class DF_branch_and_bound_opt(Displayable):
64     """returns a branch and bound searcher for a problem.
65     An optimal assignment with cost less than bound can be found by calling
           search()
66     """
67     def __init__(self, csp, bound=math.inf):
68         """creates a searcher than can be used with search() to find an
               optimal path.
69         bound gives the initial bound. By default this is infinite -
               meaning there
70         is no initial pruning due to depth bound
71         """
72         self.csp = csp
73         self.best_asst = None
74         self.bound = bound
75
76     def optimize(self):
77         """returns an optimal solution to a problem with cost less than
               bound.
78         returns None if there is no solution with cost less than bound."""
79         self.num_expanded=0
80         self.cbsearch({}, 0, self.csp.constraints)
81         self.display(1,"Number of paths expanded:",self.num_expanded)
82         return self.best_asst, self.bound
83
84     def cbsearch(self, asst, cost, constraints):
85         """finds the optimal solution that extends path and is less the
               bound"""
86         self.display(2,"cbsearch:",asst,cost,constraints)
87         can_eval = [c for c in constraints if c.can_evaluate(asst)]
88         rem_cons = [c for c in constraints if c not in can_eval]
89         newcost = cost + sum(c.value(asst) for c in can_eval)
90         self.display(2,"Evaluating:",can_eval,"cost:",newcost)
91         if newcost < self.bound:
92             self.num_expanded += 1
93             if rem_cons==[]:
94                 self.best_asst = asst
95                 self.bound = newcost
96                 self.display(1,"New best assignment:",asst," cost:",newcost)
97             else:
98                 var = next(var for var in self.csp.variables if var not in
                       asst)
```

```
99                for val in var.domain:
100                   self.cbsearch({var:val}|asst, newcost, rem_cons)
101
102 # bnb = DF_branch_and_bound_opt(scsp1)
103 # bnb.max_display_level=3 # show more detail
104 # bnb.optimize()
```

Exercise 4.17 What happens of some costs are negative? (Does it still work?) What if a value is added to all costs: does it change the optimum value, and does it affect efficiency? Make the algorithm work so that negative costs can be in the constraints. [Hint: make the smallest value be zero.]

Exercise 4.18 Change the stochastic-local search algorithms to work for soft constraints. Hint: Instead of the number of constraints violated, consider how much a change in a variable affects the objective function. Instead of returning a solution, return the best assignment found.

Chapter 5

Propositions and Inference

5.1 Representing Knowledge Bases

A clause consists of a head (an atom) and a body. A body is represented as a list of atoms. Atoms are represented as strings, or any type that can be converted to strings.

logicProblem.py — Representations Logics

```
11 class Clause(object):
12     """A definite clause"""
13
14     def __init__(self,head,body=[]):
15         """clause with atom head and lost of atoms body"""
16         self.head=head
17         self.body = body
18
19     def __repr__(self):
20         """returns the string representation of a clause.
21         """
22         if self.body:
23             return f"{self.head} <- {' & '.join(str(a) for a in
                   self.body)}."
24         else:
25             return f"{self.head}."
```

An askable atom can be asked of the user. The user can respond in English or French or just with a "y".

logicProblem.py — (continued)

```
27 class Askable(object):
28     """An askable atom"""
29
```

```
30      def __init__(self,atom):
31          """clause with atom head and lost of atoms body"""
32          self.atom=atom
33
34      def __str__(self):
35          """returns the string representation of a clause."""
36          return f"askable {self.atom}."
37
38  def yes(ans):
39      """returns true if the answer is yes in some form"""
40      return ans.lower() in ['yes', 'oui', 'y'] # bilingual
```

A knowledge base is a list of clauses and askables. To make top-down inference faster, this creates an `atom_to_clause` dictionary that maps each atom into the set of clauses with that atom in the head.

logicProblem.py — (continued)

```
42  from display import Displayable
43
44  class KB(Displayable):
45      """A knowledge base consists of a set of clauses.
46      This also creates a dictionary to give fast access to the clauses with
            an atom in head.
47      """
48      def __init__(self, statements=[]):
49          self.statements = statements
50          self.clauses = [c for c in statements if isinstance(c, Clause)]
51          self.askables = [c.atom for c in statements if isinstance(c,
                Askable)]
52          self.atom_to_clauses = {} # dictionary giving clauses with atom as
                head
53          for c in self.clauses:
54              self.add_clause(c)
55
56      def add_clause(self, c):
57          if c.head in self.atom_to_clauses:
58              self.atom_to_clauses[c.head].append(c)
59          else:
60              self.atom_to_clauses[c.head] = [c]
61
62      def clauses_for_atom(self,a):
63          """returns list of clauses with atom a as the head"""
64          if a in self.atom_to_clauses:
65              return self.atom_to_clauses[a]
66          else:
67              return []
68
69      def __str__(self):
70          """returns a string representation of this knowledge base.
71          """
72          return '\n'.join([str(c) for c in self.statements])
```

Here is a trivial example (I think therefore I am) used in the unit tests:

logicProblem.py — (continued)

```
triv_KB = KB([
    Clause('i_am', ['i_think']),
    Clause('i_think'),
    Clause('i_smell', ['i_exist'])
    ])
```

Here is a representation of the electrical domain of the textbook:

logicProblem.py — (continued)

```
elect = KB([
    Clause('light_l1'),
    Clause('light_l2'),
    Clause('ok_l1'),
    Clause('ok_l2'),
    Clause('ok_cb1'),
    Clause('ok_cb2'),
    Clause('live_outside'),
    Clause('live_l1', ['live_w0']),
    Clause('live_w0', ['up_s2','live_w1']),
    Clause('live_w0', ['down_s2','live_w2']),
    Clause('live_w1', ['up_s1', 'live_w3']),
    Clause('live_w2', ['down_s1','live_w3' ]),
    Clause('live_l2', ['live_w4']),
    Clause('live_w4', ['up_s3','live_w3' ]),
    Clause('live_p_1', ['live_w3']),
    Clause('live_w3', ['live_w5', 'ok_cb1']),
    Clause('live_p_2', ['live_w6']),
    Clause('live_w6', ['live_w5', 'ok_cb2']),
    Clause('live_w5', ['live_outside']),
    Clause('lit_l1', ['light_l1', 'live_l1', 'ok_l1']),
    Clause('lit_l2', ['light_l2', 'live_l2', 'ok_l2']),
    Askable('up_s1'),
    Askable('down_s1'),
    Askable('up_s2'),
    Askable('down_s2'),
    Askable('up_s3'),
    Askable('down_s2')
    ])

# print(kb)
```

The following knowledge base is false in the intended interpretation. One of the clauses is wrong; can you see which one? We will show how to debug it.

logicProblem.py — (continued)

```
elect_bug = KB([
    Clause('light_l2'),
    Clause('ok_l1'),
    Clause('ok_l2'),
```

```
    Clause('ok_cb1'),
    Clause('ok_cb2'),
    Clause('live_outside'),
    Clause('live_p_2', ['live_w6']),
    Clause('live_w6', ['live_w5', 'ok_cb2']),
    Clause('light_l1'),
    Clause('live_w5', ['live_outside']),
    Clause('lit_l1', ['light_l1', 'live_l1', 'ok_l1']),
    Clause('lit_l2', ['light_l2', 'live_l2', 'ok_l2']),
    Clause('live_l1', ['live_w0']),
    Clause('live_w0', ['up_s2','live_w1']),
    Clause('live_w0', ['down_s2','live_w2']),
    Clause('live_w1', ['up_s3', 'live_w3']),
    Clause('live_w2', ['down_s1','live_w3' ]),
    Clause('live_l2', ['live_w4']),
    Clause('live_w4', ['up_s3','live_w3' ]),
    Clause('live_p_1', ['live_w3']),
    Clause('live_w3', ['live_w5', 'ok_cb1']),
    Askable('up_s1'),
    Askable('down_s1'),
    Askable('up_s2'),
    Clause('light_l2'),
    Clause('ok_l1'),
    Clause('light_l2'),
    Clause('ok_l1'),
    Clause('ok_l2'),
    Clause('ok_cb1'),
    Clause('ok_cb2'),
    Clause('live_outside'),
    Clause('live_p_2', ['live_w6']),
    Clause('live_w6', ['live_w5', 'ok_cb2']),
    Clause('ok_l2'),
    Clause('ok_cb1'),
    Clause('ok_cb2'),
    Clause('live_outside'),
    Clause('live_p_2', ['live_w6']),
    Clause('live_w6', ['live_w5', 'ok_cb2']),
    Askable('down_s2'),
    Askable('up_s3'),
    Askable('down_s2')
    ])

# print(kb)
```

5.2 Bottom-up Proofs (with askables)

`fixed_point{kb}` computes the fixed point of the knowledge base `kb`.

logicBottomUp.py — Bottom-up Proof Procedure for Definite Clauses

```
from logicProblem import yes

def fixed_point(kb):
    """Returns the fixed point of knowledge base kb.
    """
    fp = ask_askables(kb)
    added = True
    while added:
        added = False  # added is true when an atom was added to fp this
            iteration
        for c in kb.clauses:
            if c.head not in fp and all(b in fp for b in c.body):
                fp.add(c.head)
                added = True
                kb.display(2,c.head,"added to fp due to clause",c)
    return fp

def ask_askables(kb):
    return {at for at in kb.askables if yes(input("Is "+at+" true? "))}
```

The following provides a trivial **unit test**, by default using the knowledge base `triv_KB`:

logicBottomUp.py — (continued)

```
from logicProblem import triv_KB
def test(kb=triv_KB, fixedpt = {'i_am','i_think'}):
    fp = fixed_point(kb)
    assert fp == fixedpt, f"kb gave result {fp}"
    print("Passed unit test")
if __name__ == "__main__":
    test()

from logicProblem import elect
# elect.max_display_level=3 # give detailed trace
# fixed_point(elect)
```

Exercise 5.1 It is not very user-friendly to ask all of the askables up-front. Implement ask-the-user so that questions are only asked if useful, and are not re-asked. For example, if there is a clause $h \leftarrow a \wedge b \wedge c \wedge d \wedge e$, where c and e are askable, c and e only need to be asked if a, b, d are all in fp and they have not been asked before. Askable e only needs to be asked if the user says "yes" to c. Askable c doesn't need to be asked if the user previously replied "no" to e, unless it is needed for some other clause.

This form of ask-the-user can ask a different set of questions than the top-down interpreter that asks questions when encountered. Give an example where they ask different questions (neither set of questions asked is a subset of the other).

Exercise 5.2 This algorithm runs in time $O(n^2)$, where n is the number of clauses, for a bounded number of elements in the body; each iteration goes through each of the clauses, and in the worst case, it will do an iteration for each clause. It is possible to implement this in time $O(n)$ time by creating an index that maps an

atom to the set of clauses with that atom in the body. Implement this. What is its complexity as a function of n and b, the maximum number of atoms in the body of a clause?

Exercise 5.3 It is possible to be more efficient (in terms of the number of elements in a body) than the method in the previous question by noticing that each element of the body of clause only needs to be checked once. For example, the clause $a \leftarrow b \wedge c \wedge d$, needs only be considered when b is added to fp. Once b is added to fp, if c is already in fp, we know that a can be added as soon as d is added. Implement this. What is its complexity as a function of n and b, the maximum number of atoms in the body of a clause?

5.3 Top-down Proofs (with askables)

The following implements the top-down proof procedure for propositional definite clauses, as described in Section 5.3.2 and Figure 5.4 of Poole and Mackworth [2023]. It implements "choose" by looping over the alternatives (using Python's any) and returning true if any choice leads to a proof.

prove(*kb*, *goal*) is used to prove *goal* from a knowledge base, *kb*, where a *goal* is a list of atoms. It returns *True* if $kb \vdash goal$. The *indent* is used when displaying the code (and doesn't need to be called initially with a non-default value).

logicTopDown.py — Top-down Proof Procedure for Definite Clauses

```
from logicProblem import yes

def prove(kb, ans_body, indent=""):
    """returns True if kb |- ans_body
    ans_body is a list of atoms to be proved
    """
    kb.display(2,indent,'yes <-',' & '.join(ans_body))
    if ans_body:
        selected = ans_body[0] # select first atom from ans_body
        if selected in kb.askables:
            return (yes(input("Is "+selected+" true? "))
                    and  prove(kb,ans_body[1:],indent+" "))
        else:
            return any(prove(kb,cl.body+ans_body[1:],indent+" ")
                       for cl in kb.clauses_for_atom(selected))
    else:
        return True  # empty body is true
```

The following provides a simple **unit test** that is hard wired for `triv_KB`:

logicTopDown.py — (continued)

```
from logicProblem import triv_KB
def test():
    a1 = prove(triv_KB,['i_am'])
    assert a1, f"triv_KB proving i_am gave {a1}"
    a2 = prove(triv_KB,['i_smell'])
```

```
    assert not a2, f"triv_KB proving i_smell gave {a2}"
    print("Passed unit tests")
if __name__ == "__main__":
    test()
# try
from logicProblem import elect
# elect.max_display_level=3 # give detailed trace
# prove(elect,['live_w6'])
# prove(elect,['lit_l1'])
```

Exercise 5.4 This code can re-ask a question multiple times. Implement this code so that it only asks a question once and remembers the answer. Also implement a function to forget the answers, which is useful if someone given an incorrect response.

Exercise 5.5 What search method is this using? Implement the search interface so that it can use A^* or other searching methods. Define an admissible heuristic that is not always 0.

5.4 Debugging and Explanation

Here we modify the top-down procedure to build a proof tree than can be traversed for explanation and debugging.

`prove_atom(kb,atom)` returns a proof for *atom* from a knowledge base *kb*, where a proof is a pair of the atom and the proofs for the elements of the body of the clause used to prove the atom. `prove_body(kb,body)` returns a list of proofs for list *body* from a knowledge base, *kb*. The *indent* is used when displaying the code (and doesn't need to have a non-default value).

logicExplain.py — Explaining Proof Procedure for Definite Clauses

```
from logicProblem import yes # for asking the user

def prove_atom(kb, atom, indent=""):
    """returns a pair (atom,proofs) where proofs is the list of proofs
       of the elements of a body of a clause used to prove atom.
    """
    kb.display(2,indent,'proving',atom)
    if atom in kb.askables:
        if yes(input("Is "+atom+" true? ")):
            return (atom,"answered")
        else:
            return "fail"
    else:
        for cl in kb.clauses_for_atom(atom):
            kb.display(2,indent,"trying",atom,'<-',' & '.join(cl.body))
            pr_body = prove_body(kb, cl.body, indent)
            if pr_body != "fail":
                return (atom, pr_body)
        return "fail"
```

```
30
31 def prove_body(kb, ans_body, indent=""):
32     """returns proof tree if kb |- ans_body or "fail" if there is no proof
33     ans_body is a list of atoms in a body to be proved
34     """
35     proofs = []
36     for atom in ans_body:
37         proof_at = prove_atom(kb, atom, indent+" ")
38         if proof_at == "fail":
39             return "fail" # fail if any proof fails
40         else:
41             proofs.append(proof_at)
42     return proofs
```

The following provides a simple **unit test** that is hard wired for `triv_KB`:

logicExplain.py — (continued)

```
44 from logicProblem import triv_KB
45 def test():
46     a1 = prove_atom(triv_KB,'i_am')
47     assert a1, f"triv_KB proving i_am gave {a1}"
48     a2 = prove_atom(triv_KB,'i_smell')
49     assert a2=="fail", "triv_KB proving i_smell gave {a2}"
50     print("Passed unit tests")
51
52 if __name__ == "__main__":
53     test()
54
55 # try
56 from logicProblem import elect, elect_bug
57 # elect.max_display_level=3 # give detailed trace
58 # prove_atom(elect, 'live_w6')
59 # prove_atom(elect, 'lit_l1')
```

The `interact(kb)` provides an interactive interface to explore proofs for knowledge base `kb`. The user can ask to prove atoms and can ask `how` an atom was proved.

To ask `how`, there must be a current atom for which there is a proof. This starts as the atom asked. When the user asks "how n" the current atom becomes the n-th element of the body of the clause used to prove the (previous) current atom. The command "up" makes the current atom the atom in the head of the rule containing the (previous) current atom. Thus "how n" moves down the proof tree and "up" moves up the proof tree, allowing the user to explore the full proof.

logicExplain.py — (continued)

```
61 helptext = """Commands are:
62 ask atom    ask is there is a proof for atom (atom should not be in quotes)
63 how         show the clause that was used to prove atom
64 how n       show the clause used to prove the nth element of the body
```

```
up          go back up proof tree to explore other parts of the proof tree
kb          print the knowledge base
quit        quit this interaction (and go back to Python)
help        print this text
"""

def interact(kb):
    going = True
    ups = []   # stack for going up
    proof="fail" # there is no proof to start
    while going:
        inp = input("logicExplain: ")
        inps = inp.split(" ")
        try:
            command = inps[0]
            if command == "quit":
                going = False
            elif command == "ask":
                proof = prove_atom(kb, inps[1])
                if proof == "fail":
                    print("fail")
                else:
                    print("yes")
            elif command == "how":
                if proof=="fail":
                    print("there is no proof")
                elif len(inps)==1:
                   print_rule(proof)
                else:
                    try:
                        ups.append(proof)
                        proof = proof[1][int(inps[1])] #nth argument of rule
                        print_rule(proof)
                    except:
                        print('In "how n", n must be a number between 0
                            and',len(proof[1])-1,"inclusive.")
            elif command == "up":
                if ups:
                    proof = ups.pop()
                else:
                    print("No rule to go up to.")
                print_rule(proof)
            elif command == "kb":
                 print(kb)
            elif command == "help":
                print(helptext)
            else:
                print("unknown command:", inp)
                print("use help for help")
        except:
```

```
            print("unknown command:", inp)
            print("use help for help")

def print_rule(proof):
    (head,body) = proof
    if body == "answered":
        print(head,"was answered yes")
    elif body == []:
             print(head,"is a fact")
    else:
            print(head,"<-")
            for i,a in enumerate(body):
                print(i,":",a[0])

# try
# interact(elect)
# Which clause is wrong in elect_bug? Try:
# interact(elect_bug)
# logicExplain: ask lit_l1
```

The following shows an interaction for the knowledge base `elect`:

```
>>> interact(elect)
logicExplain: ask lit_l1
Is up_s2 true? no
Is down_s2 true? yes
Is down_s1 true? yes
yes
logicExplain: how
lit_l1 <-
0 : light_l1
1 : live_l1
2 : ok_l1
logicExplain: how 1
live_l1 <-
0 : live_w0
logicExplain: how 0
live_w0 <-
0 : down_s2
1 : live_w2
logicExplain: how 0
down_s2 was answered yes
logicExplain: up
live_w0 <-
0 : down_s2
1 : live_w2
logicExplain: how 1
live_w2 <-
```

```
0 : down_s1
1 : live_w3
logicExplain: quit
>>>
```

Exercise 5.6 The above code only ever explores one proof – the first proof found. Change the code to enumerate the proof trees (by returning a list of all proof trees, or, preferably, using `yield`). Add the command "retry" to the user interface to try another proof.

5.5 Assumables

Atom a can be made assumable by including $Assumable(a)$ in the knowledge base. A knowledge base that can include assumables is declared with *KBA*.

logicAssumables.py — Definite clauses with assumables

```
from logicProblem import Clause, Askable, KB, yes

class Assumable(object):
    """An askable atom"""

    def __init__(self,atom):
        """clause with atom head and lost of atoms body"""
        self.atom = atom

    def __str__(self):
        """returns the string representation of a clause.
        """
        return "assumable " + self.atom + "."

class KBA(KB):
    """A knowledge base that can include assumables"""
    def __init__(self,statements):
        self.assumables = [c.atom for c in statements if isinstance(c,
            Assumable)]
        KB.__init__(self,statements)
```

The top-down Horn clause interpreter, `prove_all_ass` returns a list of the sets of assumables that imply ans_body. This list will contain all of the minimal sets of assumables, but can also find non-minimal sets, and repeated sets, if they can be generated with separate proofs. The set *assumed* is the set of assumables already assumed.

logicAssumables.py — (continued)

```
    def prove_all_ass(self, ans_body, assumed=set()):
        """returns a list of sets of assumables that extends assumed
        to imply ans_body from self.
        ans_body is a list of atoms (it is the body of the answer clause).
        assumed is a set of assumables already assumed
```

```
        """
        if ans_body:
            selected = ans_body[0] # select first atom from ans_body
            if selected in self.askables:
                if yes(input("Is "+selected+" true? ")):
                    return self.prove_all_ass(ans_body[1:],assumed)
                else:
                    return []  # no answers
            elif selected in self.assumables:
                return self.prove_all_ass(ans_body[1:],assumed|{selected})
            else:
                return [ass
                        for cl in self.clauses_for_atom(selected)
                        for ass in
                            self.prove_all_ass(cl.body+ans_body[1:],assumed)
                       ]  # union of answers for each clause with
                           head=selected
        else:               # empty body
            return [assumed]  # one answer

    def conflicts(self):
        """returns a list of minimal conflicts"""
        return minsets(self.prove_all_ass(['false']))
```

Given a list of sets, *minsets* returns a list of the minimal sets in the list. For example, *minsets*($[\{2,3,4\},\{2,3\},\{6,2,3\},\{2,3\},\{2,4,5\}]$) returns $[\{2,3\},\{2,4,5\}]$.

logicAssumables.py — (continued)

```
def minsets(ls):
    """ls is a list of sets
    returns a list of minimal sets in ls
    """
    ans = []    # elements known to be minimal
    for c in ls:
        if not any(c1<c for c1 in ls) and not any(c1 <= c for c1 in ans):
            ans.append(c)
    return ans

# minsets([{2, 3, 4}, {2, 3}, {6, 2, 3}, {2, 3}, {2, 4, 5}])
```

Warning: *minsets* works for a list of sets or for a set of (frozen) sets, but it does not work for a generator of sets (because variable `ls` is referenced in the loop). For example, try to predict and then test:

```
minsets(e for e in [{2, 3, 4}, {2, 3}, {6, 2, 3}, {2, 3}, {2, 4, 5}])
```

The diagnoses can be constructed from the (minimal) conflicts as follows. This also works if there are non-minimal conflicts, but is not as efficient.

logicAssumables.py — (continued)

```
def diagnoses(cons):
    """cons is a list of (minimal) conflicts.
```

```
    returns a list of diagnoses."""
    if cons == []:
        return [set()]
    else:
        return minsets([{e}|d)            # | is set union
                       for e in cons[0]
                       for d in diagnoses(cons[1:])])
```

Test cases:

logicAssumables.py — (continued)

```
electa = KBA([
    Clause('light_l1'),
    Clause('light_l2'),
    Assumable('ok_l1'),
    Assumable('ok_l2'),
    Assumable('ok_s1'),
    Assumable('ok_s2'),
    Assumable('ok_s3'),
    Assumable('ok_cb1'),
    Assumable('ok_cb2'),
    Assumable('live_outside'),
    Clause('live_l1', ['live_w0']),
    Clause('live_w0', ['up_s2','ok_s2','live_w1']),
    Clause('live_w0', ['down_s2','ok_s2','live_w2']),
    Clause('live_w1', ['up_s1', 'ok_s1', 'live_w3']),
    Clause('live_w2', ['down_s1', 'ok_s1','live_w3' ]),
    Clause('live_l2', ['live_w4']),
    Clause('live_w4', ['up_s3','ok_s3','live_w3' ]),
    Clause('live_p_1', ['live_w3']),
    Clause('live_w3', ['live_w5', 'ok_cb1']),
    Clause('live_p_2', ['live_w6']),
    Clause('live_w6', ['live_w5', 'ok_cb2']),
    Clause('live_w5', ['live_outside']),
    Clause('lit_l1', ['light_l1', 'live_l1', 'ok_l1']),
    Clause('lit_l2', ['light_l2', 'live_l2', 'ok_l2']),
    Askable('up_s1'),
    Askable('down_s1'),
    Askable('up_s2'),
    Askable('down_s2'),
    Askable('up_s3'),
    Askable('down_s2'),
    Askable('dark_l1'),
    Askable('dark_l2'),
    Clause('false', ['dark_l1', 'lit_l1']),
    Clause('false', ['dark_l2', 'lit_l2'])
    ])
# electa.prove_all_ass(['false'])
# cs=electa.conflicts()
# print(cs)
# diagnoses(cs)      # diagnoses from conflicts
```

Exercise 5.7 To implement a version of `conflicts` that never generates non-minimal conflicts, modify `prove_all_ass` to implement iterative deepening on the number of assumables used in a proof, and prune any set of assumables that is a superset of a conflict.

Exercise 5.8 Implement `explanations(self,body)`, where `body` is a list of atoms, that returns a list of the minimal explanations of the body. This does not require modification of `prove_all_ass`.

Exercise 5.9 Implement `explanations`, as in the previous question, so that it never generates non-minimal explanations. Hint: modify `prove_all_ass` to implement iterative deepening on the number of assumptions, generating conflicts and explanations together, and pruning as early as possible.

5.6 Negation-as-failure

The negation of an atom a is written as `Not(a)` in a body.

logicNegation.py — Propositional negation-as-failure

```
from logicProblem import KB, Clause, Askable, yes

class Not(object):
    def __init__(self, atom):
        self.theatom = atom

    def atom(self):
        return self.theatom

    def __repr__(self):
        return f"Not({self.theatom})"
```

Prove with negation-as-failure (`prove_naf`) is like prove, but with the extra case to cover `Not`:

logicNegation.py — (continued)

```
def prove_naf(kb, ans_body, indent=""):
    """ prove with negation-as-failure and askables
    returns True if kb |- ans_body
    ans_body is a list of atoms to be proved
    """
    kb.display(2,indent,'yes <-',' & '.join(str(e) for e in ans_body))
    if ans_body:
        selected = ans_body[0] # select first atom from ans_body
        if isinstance(selected, Not):
            kb.display(2,indent,f"proving {selected.atom()}")
            if prove_naf(kb, [selected.atom()], indent):
                kb.display(2,indent,f"{selected.atom()} succeeded so
                    Not({selected.atom()}) fails")
                return False
            else:
```

```
37                kb.display(2,indent,f"{selected.atom()} fails so
                      Not({selected.atom()}) succeeds")
38                return prove_naf(kb, ans_body[1:],indent+" ")
39            if selected in kb.askables:
40                return (yes(input("Is "+selected+" true? "))
41                       and  prove_naf(kb,ans_body[1:],indent+" "))
42            else:
43                return any(prove_naf(kb,cl.body+ans_body[1:],indent+" ")
44                           for cl in kb.clauses_for_atom(selected))
45        else:
46            return True   # empty body is true
```

Test cases:

logicNegation.py — (continued)

```
48 triv_KB_naf = KB([
49     Clause('i_am', ['i_think']),
50     Clause('i_think'),
51     Clause('i_smell', ['i_am', Not('dead')]),
52     Clause('i_bad', ['i_am', Not('i_think')])
53     ])
54
55 triv_KB_naf.max_display_level = 4
56 def test():
57     a1 = prove_naf(triv_KB_naf,['i_smell'])
58     assert a1, f"triv_KB_naf failed to prove i_smell; gave {a1}"
59     a2 = prove_naf(triv_KB_naf,['i_bad'])
60     assert not a2, f"triv_KB_naf wrongly proved i_bad; gave {a2}"
61     print("Passed unit tests")
62 if __name__ == "__main__":
63     test()
```

Default reasoning about beaches at resorts (Example 5.28 of Poole and Mackworth [2023]):

logicNegation.py — (continued)

```
65 beach_KB = KB([
66    Clause('away_from_beach', [Not('on_beach')]),
67    Clause('beach_access', ['on_beach', Not('ab_beach_access')]),
68    Clause('swim_at_beach', ['beach_access', Not('ab_swim_at_beach')]),
69    Clause('ab_swim_at_beach', ['enclosed_bay', 'big_city',
          Not('ab_no_swimming_near_city')]),
70    Clause('ab_no_swimming_near_city', ['in_BC', Not('ab_BC_beaches')])
71     ])
72
73 # prove_naf(beach_KB, ['away_from_beach'])
74 # prove_naf(beach_KB, ['beach_access'])
75 # beach_KB.add_clause(Clause('on_beach',[]))
76 # prove_naf(beach_KB, ['away_from_beach'])
77 # prove_naf(beach_KB, ['swim_at_beach'])
78 # beach_KB.add_clause(Clause('enclosed_bay',[]))
79 # prove_naf(beach_KB, ['swim_at_beach'])
```

```
# beach_KB.add_clause(Clause('big_city',[]))
# prove_naf(beach_KB, ['swim_at_beach'])
# beach_KB.add_clause(Clause('in_BC',[]))
# prove_naf(beach_KB, ['swim_at_beach'])
```

Chapter 6

Deterministic Planning

6.1 Representing Actions and Planning Problems

The STRIPS representation of an action consists of:

- the name of the action
- preconditions: a dictionary of *feature:value* pairs that specifies that the feature must have this value for the action to be possible
- effects: a dictionary of *feature:value* pairs that are made true by this action. In particular, a feature in the dictionary has the corresponding value (and not its previous value) after the action, and a feature not in the dictionary keeps its old value.
- a cost for the action

stripsProblem.py — STRIPS Representations of Actions

```
11 class Strips(object):
12     def __init__(self, name, preconds, effects, cost=1):
13         """
14         defines the STRIPS representation for an action:
15         * name is the name of the action
16         * preconds, the preconditions, is feature:value dictionary that
               must hold
17         for the action to be carried out
18         * effects is a feature:value map that this action makes
19         true. The action changes the value of any feature specified
20         here, and leaves other features unchanged.
```

```
        * cost is the cost of the action
        """
        self.name = name
        self.preconds = preconds
        self.effects = effects
        self.cost = cost

    def __repr__(self):
        return self.name
```

A STRIPS domain consists of:

- A dictionary `feature_domain_dict` that maps each feature into a set of possible values for the feature. This is needed for the CSP planner.
- A set of actions, each represented using the `Strips` class.

stripsProblem.py — (continued)

```
class STRIPS_domain(object):
    def __init__(self, feature_domain_dict, actions):
        """Problem domain
        feature_domain_dict is a feature:domain dictionary,
               mapping each feature to its domain
        actions
        """
        self.feature_domain_dict = feature_domain_dict
        self.actions = actions
```

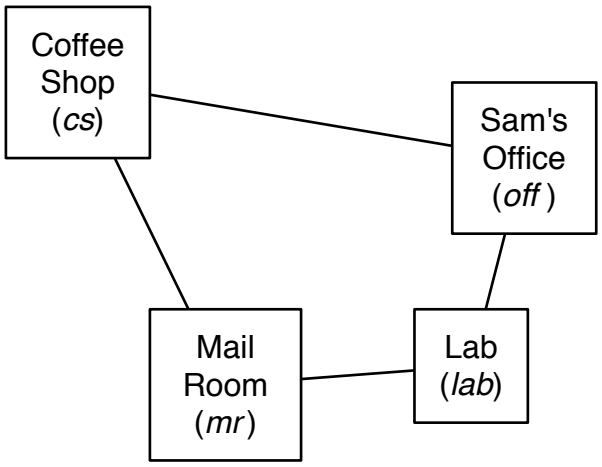
A planning problem consists of a planning domain, an initial state, and a goal. The goal does not need to fully specify the final state.

stripsProblem.py — (continued)

```
class Planning_problem(object):
    def __init__(self, prob_domain, initial_state, goal):
        """
        a planning problem consists of
        * a planning domain
        * the initial state
        * a goal
        """
        self.prob_domain = prob_domain
        self.initial_state = initial_state
        self.goal = goal
```

6.1.1 Robot Delivery Domain

The following specifies the robot delivery domain of Section 6.1, shown in Figure 6.1.



Features to describe states

RLoc – Rob's location
RHC – Rob has coffee
SWC – Sam wants coffee
MW – Mail is waiting
RHM – Rob has mail

Actions

mc – move clockwise
mcc – move counterclockwise
puc – pickup coffee
dc – deliver coffee
pum – pickup mail
dm – deliver mail

Figure 6.1: Robot Delivery Domain

stripsProblem.py — (continued)

```
boolean = {False, True}
delivery_domain = STRIPS_domain(
   {'RLoc':{'cs', 'off', 'lab', 'mr'}, 'RHC':boolean, 'SWC':boolean,
    'MW':boolean, 'RHM':boolean},       #feature:values dictionary
   { Strips('mc_cs', {'RLoc':'cs'}, {'RLoc':'off'}),
    Strips('mc_off', {'RLoc':'off'}, {'RLoc':'lab'}),
    Strips('mc_lab', {'RLoc':'lab'}, {'RLoc':'mr'}),
    Strips('mc_mr', {'RLoc':'mr'}, {'RLoc':'cs'}),
    Strips('mcc_cs', {'RLoc':'cs'}, {'RLoc':'mr'}),
    Strips('mcc_off', {'RLoc':'off'}, {'RLoc':'cs'}),
    Strips('mcc_lab', {'RLoc':'lab'}, {'RLoc':'off'}),
    Strips('mcc_mr', {'RLoc':'mr'}, {'RLoc':'lab'}),
    Strips('puc', {'RLoc':'cs', 'RHC':False}, {'RHC':True}),
    Strips('dc', {'RLoc':'off', 'RHC':True}, {'RHC':False, 'SWC':False}),
    Strips('pum', {'RLoc':'mr','MW':True}, {'RHM':True,'MW':False}),
    Strips('dm', {'RLoc':'off', 'RHM':True}, {'RHM':False})
   } )
```

stripsProblem.py — (continued)

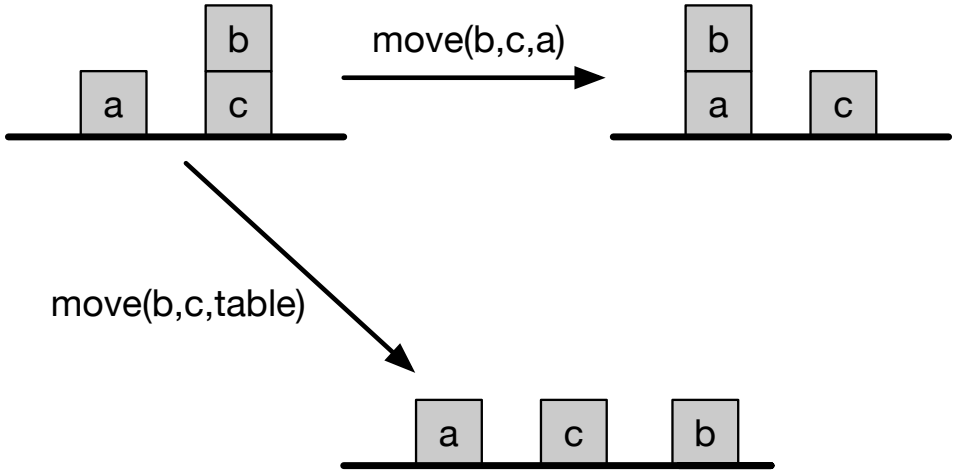


Figure 6.2: Blocks world with two actions

```
problem0 = Planning_problem(delivery_domain,
                            {'RLoc':'lab', 'MW':True, 'SWC':True, 'RHC':False,
                             'RHM':False},
                            {'RLoc':'off'})
problem1 = Planning_problem(delivery_domain,
                            {'RLoc':'lab', 'MW':True, 'SWC':True, 'RHC':False,
                             'RHM':False},
                            {'SWC':False})
problem2 = Planning_problem(delivery_domain,
                            {'RLoc':'lab', 'MW':True, 'SWC':True, 'RHC':False,
                             'RHM':False},
                            {'SWC':False, 'MW':False, 'RHM':False})
```

6.1.2 Blocks World

The blocks world consist of blocks and a table. Each block can be on the table or on another block. A block can only have one other block on top of it. Figure 6.2 shows 3 states with some of the actions between them.

A state is defined by the two features:

- *on* where $on(x) = y$ when block x is on block or table y
- *clear* where $clear(x) = True$ when block x has nothing on it.

There is one parameterized action

- $move(x, y, z)$ move block x from y to z, where y and z could be a block or the table.

To handle parameterized actions (which depend on the blocks involved), the actions and the features are all strings, created for all the combinations of the blocks. Note that we treat moving to a block separately from moving to the table, because the blocks needs to be clear, but the table always has room for another block.

stripsProblem.py — (continued)

```
### blocks world
def move(x,y,z):
    """string for the 'move' action"""
    return 'move_'+x+'_from_'+y+'_to_'+z
def on(x):
    """string for the 'on' feature"""
    return x+'_is_on'
def clear(x):
    """string for the 'clear' feature"""
    return 'clear_'+x
def create_blocks_world(blocks = {'a','b','c','d'}):
    blocks_and_table = blocks | {'table'}
    stmap = {Strips(move(x,y,z),{on(x):y, clear(x):True, clear(z):True},
                                {on(x):z, clear(y):True, clear(z):False})
                    for x in blocks
                    for y in blocks_and_table
                    for z in blocks
                    if x!=y and y!=z and z!=x}
    stmap.update({Strips(move(x,y,'table'), {on(x):y, clear(x):True},
                                {on(x):'table', clear(y):True})
                    for x in blocks
                    for y in blocks
                    if x!=y})
    feature_domain_dict = {on(x):blocks_and_table-{x} for x in blocks}
    feature_domain_dict.update({clear(x):boolean for x in blocks_and_table})
    return STRIPS_domain(feature_domain_dict, stmap)
```

The problem *blocks*1 is a classic example, with 3 blocks, and the goal consists of two conditions. See Figure 6.3. This example is challenging because you can't achieve one of the goals (using the minimum number of actions) and then the other; whichever one you achieve first has to be undone to achieve the second.

stripsProblem.py — (continued)

```
blocks1dom = create_blocks_world({'a','b','c'})
blocks1 = Planning_problem(blocks1dom,
    {on('a'):'table', clear('a'):True,
     on('b'):'c', clear('b'):True,
     on('c'):'table', clear('c'):False}, # initial state
    {on('a'):'b', on('c'):'a'}) #goal
```

The problem *blocks*2 is one to invert a tower of size 4.

stripsProblem.py — (continued)

```
blocks2dom = create_blocks_world({'a','b','c','d'})
```

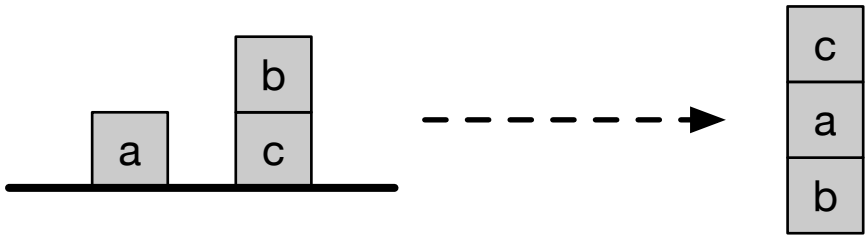


Figure 6.3: Blocks problem blocks1

```
tower4 = {clear('a'):True, on('a'):'b',
         clear('b'):False, on('b'):'c',
         clear('c'):False, on('c'):'d',
         clear('d'):False, on('d'):'table'}
blocks2 = Planning_problem(blocks2dom,
    tower4, # initial state
    {on('d'):'c',on('c'):'b',on('b'):'a'}) #goal
```

The problem *blocks*3 is to move the bottom block to the top of a tower of size 4.

stripsProblem.py — (continued)

```
blocks3 = Planning_problem(blocks2dom,
    tower4, # initial state
    {on('d'):'a', on('a'):'b', on('b'):'c'}) #goal
```

Exercise 6.1 Represent the problem of given a tower of 4 blocks (a on b on c on d on table), the goal is to have a tower with the previous top block on the bottom (b on c on d on a). Do not include the table in your goal (the goal does not care whether a is on the table). [Before you run the program, estimate how many steps it will take to solve this.] How many steps does an optimal planner take?

Exercise 6.2 Represent the domain so that $on(x, y)$ is a Boolean feature that is True when x is on y, Does the representation of the state need to include negative *on* facts? Why or why not? (Note that this may depend on the planner; write your answer with respect to particular planners.)

Exercise 6.3 It is possible to write the representation of the problem without using *clear*, where $clear(x)$ means nothing is on x. Change the definition of the blocks world so that it does not use *clear* but uses *on* being false instead. Does this work better for any of the planners?

6.2 Forward Planning

To run the demo, in folder "aipython", load "stripsForwardPlanner.py", and copy and paste the commented-out example queries at the bottom of that file.

In a forward planner, a node is a state. A state consists of an assignment, a feature:value dictionary, where all features have a value. Multiple-path pruning requires a hash function, and equality between states.

stripsForwardPlanner.py — Forward Planner with STRIPS actions

```
from searchProblem import Arc, Search_problem
from stripsProblem import Strips, STRIPS_domain

class State(object):
    def __init__(self,assignment):
        self.assignment = assignment
        self.hash_value = None
    def __hash__(self):
        if self.hash_value is None:
            self.hash_value = hash(frozenset(self.assignment.items()))
        return self.hash_value
    def __eq__(self,st):
        return self.assignment == st.assignment
    def __str__(self):
        return str(self.assignment)
```

To define a search problem (page 41), you need to define the goal condition, the start nodes, the neighbors, and (optionally) a heuristic function. Here `zero` is the default heuristic function.

stripsForwardPlanner.py — (continued)

```
def zero(*args,**nargs):
    """always returns 0"""
    return 0

class Forward_STRIPS(Search_problem):
    """A search problem from a planning problem where:
    * a node is a state
    * the dynamics are specified by the STRIPS representation of actions
    """
    def __init__(self, planning_problem, heur=zero):
        """creates a forward search space from a planning problem.
        heur(state,goal) is a heuristic function,
           an underestimate of the cost from state to goal, where
           both state and goals are feature:value dictionaries.
        """
        self.prob_domain = planning_problem.prob_domain
        self.initial_state = State(planning_problem.initial_state)
        self.goal = planning_problem.goal
        self.heur = heur

    def is_goal(self, state):
        """is True if node is a goal.

        Every goal feature has the same value in the state and the goal."""
        return all(state.assignment[prop]==self.goal[prop]
```

```
                for prop in self.goal)

    def start_node(self):
        """returns start node"""
        return self.initial_state

    def neighbors(self,state):
        """returns neighbors of state in this problem"""
        return [ Arc(state, self.effect(act,state.assignment), act.cost,
            act)
                for act in self.prob_domain.actions
                if self.possible(act,state.assignment)]

    def possible(self,act,state_asst):
        """True if act is possible in state.
        act is possible if all of its preconditions have the same value in
            the state"""
        return all(state_asst[pre] == act.preconds[pre]
                   for pre in act.preconds)

    def effect(self,act,state_asst):
        """returns the state that is the effect of doing act given
            state_asst
        Python 3.9: return state_asst | act.effects"""
        new_state_asst = state_asst.copy()
        new_state_asst.update(act.effects)
        return State(new_state_asst)

    def heuristic(self,state):
        """in the forward planner a node is a state.
        the heuristic is an (under)estimate of the cost
        of going from the state to the top-level goal.
        """
        return self.heur(state.assignment, self.goal)
```

Here are some test cases to try.

stripsForwardPlanner.py — (continued)

```
from searchBranchAndBound import DF_branch_and_bound
from searchMPP import SearcherMPP
import stripsProblem

# SearcherMPP(Forward_STRIPS(stripsProblem.problem1)).search() #A* with MPP
# DF_branch_and_bound(Forward_STRIPS(stripsProblem.problem1),10).search()
    #B&B
# To find more than one plan:
# s1 = SearcherMPP(Forward_STRIPS(stripsProblem.problem1)) #A*
# s1.search() #find another plan
```

6.2.1 Defining Heuristics for a Planner

Each planning domain requires its own heuristics. If you change the actions, you will need to reconsider the heuristic function, as there might then be a lower-cost path, which might make the heuristic non-admissible.

Here is an example of defining heuristics for the coffee delivery planning domain.

First define the distance between two locations, which is used for the heuristics.

stripsHeuristic.py — Planner with Heuristic Function

```
11 def dist(loc1, loc2):
12     """returns the distance from location loc1 to loc2
13     """
14     if loc1==loc2:
15         return 0
16     if {loc1,loc2} in [{'cs','lab'},{'mr','off'}]:
17         return 2
18     else:
19         return 1
```

Note that the current state is a complete description; there is a value for every feature. However the goal need not be complete; it does not need to define a value for every feature. Before checking the value for a feature in the goal, a heuristic needs to define whether the feature is defined in the goal.

stripsHeuristic.py — (continued)

```
21 def h1(state,goal):
22     """ the distance to the goal location, if there is one"""
23     if 'RLoc' in goal:
24         return dist(state['RLoc'], goal['RLoc'])
25     else:
26         return 0
27
28 def h2(state,goal):
29     """ the distance to the coffee shop plus getting coffee and delivering
           it
30     if the robot needs to get coffee
31     """
32     if ('SWC' in goal and goal['SWC']==False
33             and state['SWC']==True
34             and state['RHC']==False):
35         return dist(state['RLoc'],'cs')+3
36     else:
37         return 0
```

The maximum of the values of a set of admissible heuristics is also an admissible heuristic. The function `maxh` takes a number of heuristic functions as arguments, and returns a new heuristic function that takes the maximum of the values of the heuristics. For example, `h1` and `h2` are heuristic functions and so `maxh(h1,h2)` is also. `maxh` can take an arbitrary number of arguments.

stripsHeuristic.py — (continued)

```
def maxh(*heuristics):
    """Returns a new heuristic function that is the maximum of the
        functions in heuristics.
    heuristics is the list of arguments which must be heuristic functions.
    """
    # return lambda state,goal: max(h(state,goal) for h in heuristics)
    def newh(state,goal):
        return max(h(state,goal) for h in heuristics)
    return newh
```

The following runs the example with and without the heuristic.

stripsHeuristic.py — (continued)

```
##### Forward Planner #####
from searchMPP import SearcherMPP
from stripsForwardPlanner import Forward_STRIPS
import stripsProblem

def test_forward_heuristic(thisproblem=stripsProblem.problem1):
    print("\n***** FORWARD NO HEURISTIC")
    print(SearcherMPP(Forward_STRIPS(thisproblem)).search())

    print("\n***** FORWARD WITH HEURISTIC h1")
    print(SearcherMPP(Forward_STRIPS(thisproblem,h1)).search())

    print("\n***** FORWARD WITH HEURISTIC h2")
    print(SearcherMPP(Forward_STRIPS(thisproblem,h2)).search())

    print("\n***** FORWARD WITH HEURISTICs h1 and h2")
    print(SearcherMPP(Forward_STRIPS(thisproblem,maxh(h1,h2))).search())

if __name__ == "__main__":
    test_forward_heuristic()
```

Exercise 6.4 For more than one start-state/goal combination, test the forward planner with a heuristic function of just `h1`, with just `h2` and with both. Explain why each one prunes or doesn't prune the search space.

Exercise 6.5 Create a better heuristic than `maxh(h1,h2)`. Try it for a number of different problems. In particular, try and include the following costs:

i) `h3` is like `h2` but also takes into account the case when `Rloc` is in goal.

ii) `h4` uses the distance to the mail room plus getting mail and delivering it if the robot needs to get need to deliver mail.

iii) `h5` is for getting mail when goal is for the robot to have mail, and then getting to the goal destination (if there is one).

Exercise 6.6 Create an admissible heuristic for the blocks world.

6.3 Regression Planning

> To run the demo, in folder "aipython", load "stripsRegressionPlanner.py", and copy and paste the commented-out example queries at the bottom of that file.

In a regression planner a node is a subgoal that need to be achieved. A Subgoal consists of an assignment, a *feature:value* dictionary, which assigns some – but typically not all – of the state features. It is hashable so that multiple path pruning can work. The hash is only computed when necessary (and only once).

stripsRegressionPlanner.py — Regression Planner with STRIPS actions

```
from searchProblem import Arc, Search_problem

class Subgoal(object):
    def __init__(self,assignment):
        self.assignment = assignment
        self.hash_value = None
    def __hash__(self):
        if self.hash_value is None:
            self.hash_value = hash(frozenset(self.assignment.items()))
        return self.hash_value
    def __eq__(self,st):
        return self.assignment == st.assignment
    def __str__(self):
        return str(self.assignment)
```

A regression search has subgoals as nodes. The initial node is the top-level goal of the planner. The goal for the search (when the search can stop) is a subgoal that holds in the initial state.

stripsRegressionPlanner.py — (continued)

```
from stripsForwardPlanner import zero

class Regression_STRIPS(Search_problem):
    """A search problem where:
    * a node is a goal to be achieved, represented by a set of propositions.
    * the dynamics are specified by the STRIPS representation of actions
    """

    def __init__(self, planning_problem, heur=zero):
        """creates a regression search space from a planning problem.
        heur(state,goal) is a heuristic function;
           an underestimate of the cost from state to goal, where
           both state and goals are feature:value dictionaries
        """
        self.prob_domain = planning_problem.prob_domain
        self.top_goal = Subgoal(planning_problem.goal)
        self.initial_state = planning_problem.initial_state
```

```
43          self.heur = heur
44
45      def is_goal(self, subgoal):
46          """if subgoal is true in the initial state, a path has been found"""
47          goal_asst = subgoal.assignment
48          return all(self.initial_state[g]==goal_asst[g]
49                     for g in goal_asst)
50
51      def start_node(self):
52          """the start node is the top-level goal"""
53          return self.top_goal
54
55      def neighbors(self,subgoal):
56          """returns a list of the arcs for the neighbors of subgoal in this
                problem"""
57          goal_asst = subgoal.assignment
58          return [ Arc(subgoal, self.weakest_precond(act,goal_asst),
                    act.cost, act)
59                   for act in self.prob_domain.actions
60                   if self.possible(act,goal_asst)]
61
62      def possible(self,act,goal_asst):
63          """True if act is possible to achieve goal_asst.
64
65          the action achieves an element of the effects and
66          the action doesn't delete something that needs to be achieved and
67          the preconditions are consistent with other subgoals that need to
                be achieved
68          """
69          return ( any(goal_asst[prop] == act.effects[prop]
70                       for prop in act.effects if prop in goal_asst)
71                   and all(goal_asst[prop] == act.effects[prop]
72                           for prop in act.effects if prop in goal_asst)
73                   and all(goal_asst[prop]== act.preconds[prop]
74                           for prop in act.preconds if prop not in act.effects
                                and prop in goal_asst)
75                   )
76
77      def weakest_precond(self,act,goal_asst):
78          """returns the subgoal that must be true so goal_asst holds after
                act
79          should be:  act.preconds | (goal_asst - act.effects)
80          """
81          new_asst = act.preconds.copy()
82          for g in goal_asst:
83              if g not in act.effects:
84                  new_asst[g] = goal_asst[g]
85          return Subgoal(new_asst)
86
87      def heuristic(self,subgoal):
```

```
88          """in the regression planner a node is a subgoal.
89          the heuristic is an (under)estimate of the cost of going from the
                initial state to subgoal.
90          """
91          return self.heur(self.initial_state, subgoal.assignment)
```

stripsRegressionPlanner.py — (continued)

```
93  from searchBranchAndBound import DF_branch_and_bound
94  from searchMPP import SearcherMPP
95  import stripsProblem
96
97  # SearcherMPP(Regression_STRIPS(stripsProblem.problem1)).search() #A* with
        MPP
98  #
        DF_branch_and_bound(Regression_STRIPS(stripsProblem.problem1),10).search()
        #B&B
```

Exercise 6.7 Multiple path pruning could be used to prune more than the current node. In particular, if the current node contains more conditions than a previously visited node, it can be pruned. For example, if `{a:True, b:False}` has been visited, then any node that is a superset, e.g., `{a:True, b:False, d:True}`, need not be expanded. If the simpler subgoal does not lead to a solution, the more complicated one will not either. Implement this more severe pruning. (Hint: This may require modifications to the searcher.)

Exercise 6.8 It is possible that, as knowledge of the domain, that some assignment of values to features can never be achieved. For example, the robot cannot be holding mail when there is mail waiting (assuming it isn't holding mail initially). An assignment of values to (some of the) features is incompatible if no possible (reachable) state can include that assignment. For example, `{'MW':True, 'RHM':True}` is an incompatible assignment. This information may be useful information for a planner; there is no point in trying to achieve these together. Define a subclass of `STRIPS_domain` that can accept a list of incompatible assignments. Modify the regression planner code to use such a list of incompatible assignments. Give an example where the search space is smaller.

Exercise 6.9 After completing the previous exercise, design incompatible assignments for the blocks world. (This can result in dramatic search improvements.)

6.3.1 Defining Heuristics for a Regression Planner

The regression planner can use the same heuristic function as the forward planner. However, just because a heuristic is useful for a forward planner does not mean it is useful for a regression planner, and vice versa. you should experiment with whether the same heuristic works well for both a regression planner and a forward planner.

The following runs the same example as the forward planner with and without the heuristic defined for the forward planner:

stripsHeuristic.py — (continued)

```
69 ##### Regression Planner
70 from stripsRegressionPlanner import Regression_STRIPS
71
72 def test_regression_heuristic(thisproblem=stripsProblem.problem1):
73     print("\n***** REGRESSION NO HEURISTIC")
74     print(SearcherMPP(Regression_STRIPS(thisproblem)).search())
75
76     print("\n***** REGRESSION WITH HEURISTICs h1 and h2")
77     print(SearcherMPP(Regression_STRIPS(thisproblem,maxh(h1,h2))).search())
78
79 if __name__ == "__main__":
80     test_regression_heuristic()
```

Exercise 6.10 Try the regression planner with a heuristic function of just `h1` and with just `h2` (defined in Section 6.2.1). Explain how each one prunes or doesn't prune the search space.

Exercise 6.11 Create a heuristic that is better for regression planning than `heuristic_fun` defined in Section 6.2.1.

6.4 Planning as a CSP

To run the demo, in folder "aipython", load "stripsCSPPlanner.py", and copy and paste the commented-out example queries at the bottom of that file. This assumes Python 3.

The CSP planner assumes there is a single action at each step. This creates a CSP that can use any of the CSP algorithms to solve (e.g., stochastic local search or arc consistency with domain splitting).

It uses the same action representation as before; it does not consider factored actions (action features), or implement state constraints.

stripsCSPPlanner.py — CSP planner where actions are represented using STRIPS

```
11 from cspProblem import Variable, CSP, Constraint
12
13 class CSP_from_STRIPS(CSP):
14     """A CSP where:
15     * CSP variables are constructed for each feature and time, and each
           action and time
16     * the dynamics are specified by the STRIPS representation of actions
17     """
18
19     def __init__(self, planning_problem, number_stages=2):
20         prob_domain = planning_problem.prob_domain
21         initial_state = planning_problem.initial_state
22         goal = planning_problem.goal
23         # self.action_vars[t] is the action variable for time t
```

```
        self.action_vars = [Variable(f"Action{t}", prob_domain.actions)
                            for t in range(number_stages)]
        # feat_time_var[f][t] is the variable for feature f at time t
        feat_time_var = {feat: [Variable(f"{feat}_{t}",dom)
                                for t in range(number_stages+1)]
                        for (feat,dom) in
                            prob_domain.feature_domain_dict.items()}

        # initial state constraints:
        constraints = [Constraint([feat_time_var[feat][0]], is_(val),
                                  f"{feat}[0]={val}")
                       for (feat,val) in initial_state.items()]

        # goal constraints on the final state:
        constraints += [Constraint([feat_time_var[feat][number_stages]],
                                   is_(val),
                                       f"{feat}[{number_stages}]={val}")
                        for (feat,val) in goal.items()]

        # precondition constraints:
        constraints += [Constraint([feat_time_var[feat][t],
            self.action_vars[t]],
                                   if_(val,act),
                                   f"{feat}[{t}]={val} if action[{t}]={act}")
                        for act in prob_domain.actions
                        for (feat,val) in act.preconds.items()
                        for t in range(number_stages)]

        # effect constraints:
        constraints += [Constraint([feat_time_var[feat][t+1],
            self.action_vars[t]],
                                   if_(val,act),
                                   f"{feat}[{t+1}]={val} if action[{t}]={act}")
                        for act in prob_domain.actions
                        for feat,val in act.effects.items()
                        for t in range(number_stages)]
        # frame constraints:

        constraints += [Constraint([feat_time_var[feat][t],
            self.action_vars[t], feat_time_var[feat][t+1]],
                                   eq_if_not_in_({act for act in
                                       prob_domain.actions
                                                if feat in act.effects}),
                                  f"{feat}[t]={feat}[{t+1}] if act not in
                                      {set(act for act in prob_domain.actions
                                       if feat in act.effects)}")
                        for feat in prob_domain.feature_domain_dict
                        for t in range(number_stages) ]
        variables = set(self.action_vars) | {feat_time_var[feat][t]
                                             for feat in
```

```
                                                prob_domain.feature_domain_dict
                                            for t in range(number_stages+1)}
        CSP.__init__(self, "CSP_from_Strips", variables, constraints)

    def extract_plan(self,soln):
        return [soln[a] for a in self.action_vars]
```

The following methods return methods which can be applied to the particular environment.

For example, `is_(3)` returns a function that when applied to 3, returns True and when applied to any other value returns False. So `is_(3)(3)` returns `True` and `is_(3)(7)` returns `False`.

Note that the underscore ('_') is part of the name; we use the convention that a function with name ending in underscore returns a function. Commented out is an alternative style to define `is_` and `if_`; returning a function defined by `lambda` is equivalent to returning the embedded function, except that the embedded function has a name. The embedded function can also be given a docstring.

stripsCSPPlanner.py — (continued)

```
def is_(val):
    """returns a function that is true when it is it applied to val.
    """
    #return lambda x: x == val
    def is_fun(x):
        return x == val
    is_fun.__name__ = f"value_is_{val}"
    return is_fun

def if_(v1,v2):
    """if the second argument is v2, the first argument must be v1"""
    #return lambda x1,x2: x1==v1 if x2==v2 else True
    def if_fun(x1,x2):
        return x1==v1 if x2==v2 else True
    if_fun.__name__ = f"if x2 is {v2} then x1 is {v1}"
    return if_fun

def eq_if_not_in_(actset):
    """first and third arguments are equal if action is not in actset"""
    # return lambda x1, a, x2: x1==x2 if a not in actset else True
    def eq_if_not_fun(x1, a, x2):
        return x1==x2 if a not in actset else True
    eq_if_not_fun.__name__ = f"first and third arguments are equal if
        action is not in {actset}"
    return eq_if_not_fun
```

Putting it together, this returns a list of actions that solves the problem for a given horizon. If you want to do more than just return the list of actions, you might want to get it to return the solution. Or even enumerate the solutions (by using `Search_with_AC_from_CSP`).

stripsCSPPlanner.py — (continued)

```
def con_plan(prob,horizon):
    """finds a plan for problem prob given horizon.
    """
    csp = CSP_from_STRIPS(prob, horizon)
    sol = Con_solver(csp).solve_one()
    return csp.extract_plan(sol) if sol else sol
```

The following are some example queries.

stripsCSPPlanner.py — (continued)

```
from searchGeneric import Searcher
from cspConsistency import Search_with_AC_from_CSP, Con_solver
from stripsProblem import Planning_problem
import stripsProblem

# Problem 0
# con_plan(stripsProblem.problem0,1) # should it succeed?
# con_plan(stripsProblem.problem0,2) # should it succeed?
# con_plan(stripsProblem.problem0,3) # should it succeed?
# To use search to enumerate solutions
#searcher0a =
    Searcher(Search_with_AC_from_CSP(CSP_from_STRIPS(stripsProblem.problem0,
    1)))
#print(searcher0a.search()) # returns path to solution

## Problem 1
# con_plan(stripsProblem.problem1,5) # should it succeed?
# con_plan(stripsProblem.problem1,4) # should it succeed?
## To use search to enumerate solutions:
#searcher15a =
    Searcher(Search_with_AC_from_CSP(CSP_from_STRIPS(stripsProblem.problem1,
    5)))
#print(searcher15a.search()) # returns path to solution

## Problem 2
#con_plan(stripsProblem.problem2, 6) # should fail??
#con_plan(stripsProblem.problem2, 7) # should succeed???

## Example 6.13
problem3 = Planning_problem(stripsProblem.delivery_domain,
                            {'SWC':True, 'RHC':False}, {'SWC':False})
#con_plan(problem3,2) # Horizon of 2
#con_plan(problem3,3) # Horizon of 3

problem4 = Planning_problem(stripsProblem.delivery_domain,{'SWC':True},
                              {'SWC':False, 'MW':False, 'RHM':False})

# For the stochastic local search:
#from cspSLS import SLSearcher, Runtime_distribution
```

```
# cspplanning15 = CSP_from_STRIPS(stripsProblem.problem1, 5) # should
    succeed
#se0 = SLSearcher(cspplanning15); print(se0.search(100000,0.5))
#p = Runtime_distribution(cspplanning15)
#p.plot_runs(1000,1000,0.7) # warning may take a few minutes
```

6.5 Partial-Order Planning

> To run the demo, in folder "aipython", load "stripsPOP.py", and copy and paste the commented-out example queries at the bottom of that file.

A partial order planner maintains a partial order of action instances. An action instance consists of a name and an index. You need action instances because the same action could be carried out at different times.

stripsPOP.py — Partial-order Planner using STRIPS representation

```
from searchProblem import Arc, Search_problem
import random

class Action_instance(object):
    next_index = 0
    def __init__(self,action,index=None):
        if index is None:
            index = Action_instance.next_index
            Action_instance.next_index += 1
        self.action = action
        self.index = index

    def __str__(self):
        return f"{self.action}#{self.index}"

    __repr__ = __str__ # __repr__ function is the same as the __str__
        function
```

A partial-order planner is represented as a search problem (Section 3.1) where a node consists of:

- actions: a set of action instances.
- constraints: a set of (a_1, a_2) pairs, where a_1 and a_2 are action instances, which represents that a_1 must come before a_2 in the partial order. There are a number of ways that this could be represented. The code below represents the set of pairs that are in transitive closure of the *before* relation. This lets it quickly determine whether some *before* relation is consistent with the current constraints, at the cost of pre-computing and storing the transitive closure.

- *agenda*: a list of (s, a) pairs, where s is a (var, val) pair and a is an action instance. This means that variable *var* must have value *val* before a can occur.
- *causal_links*: a set of $(a0, g, a1)$ triples, where a_1 and a_2 are action instances and g is a (var, val) pair. This holds when action a_0 makes g true for action a_1.

stripsPOP.py — (continued)

```
class POP_node(object):
    """a (partial) partial-order plan. This is a node in the search
        space."""
    def __init__(self, actions, constraints, agenda, causal_links):
        """
        * actions is a set of action instances
        * constraints a set of (a0,a1) pairs, representing a0<a1,
          closed under transitivity
        * agenda list of (subgoal,action) pairs to be achieved, where
          subgoal is a (variable,value) pair
        * causal_links is a set of (a0,g,a1) triples,
          where ai are action instances, and g is a (variable,value) pair
        """
        self.actions = actions # a set of action instances
        self.constraints = constraints # a set of (a0,a1) pairs
        self.agenda = agenda  # list of (subgoal,action) pairs to be
            achieved
        self.causal_links = causal_links # set of (a0,g,a1) triples

    def __str__(self):
        return ("actions: "+str({str(a) for a in self.actions})+
                "\nconstraints: "+
                str({(str(a1),str(a2)) for (a1,a2) in self.constraints})+
                "\nagenda: "+
                str([(str(s),str(a)) for (s,a) in self.agenda])+
                "\ncausal_links:"+
                str({(str(a0),str(g),str(a2)) for (a0,g,a2) in
                    self.causal_links}) )
```

`extract_plan` constructs a total order of action instances that is consistent with the partial order.

stripsPOP.py — (continued)

```
    def extract_plan(self):
        """returns a total ordering of the action instances consistent
        with the constraints.
        raises IndexError if there is no choice.
        """
        sorted_acts = []
        other_acts = set(self.actions)
        while other_acts:
```

```
62            a = random.choice([a for a in other_acts if
63                    all(((a1,a) not in self.constraints) for a1 in
                          other_acts)])
64            sorted_acts.append(a)
65            other_acts.remove(a)
66        return sorted_acts
```

POP_search_from_STRIPS is an instance of a search problem. As such, it needs start nodes, a goal, and the neighbors function.

stripsPOP.py — (continued)

```
68 from display import Displayable
69
70 class POP_search_from_STRIPS(Search_problem, Displayable):
71     def __init__(self,planning_problem):
72         Search_problem.__init__(self)
73         self.planning_problem = planning_problem
74         self.start = Action_instance("start")
75         self.finish = Action_instance("finish")
76
77     def is_goal(self, node):
78         return node.agenda == []
79
80     def start_node(self):
81         constraints = {(self.start, self.finish)}
82         agenda = [(g, self.finish) for g in
                      self.planning_problem.goal.items()]
83         return POP_node([self.start,self.finish], constraints, agenda, [] )
```

The neighbors method enumerates the neighbors of a given node, using yield.

stripsPOP.py — (continued)

```
85     def neighbors(self, node):
86         """enumerates the neighbors of node"""
87         self.display(3,"finding neighbors of\n",node)
88         if node.agenda:
89             subgoal,act1 = node.agenda[0]
90             self.display(2,"selecting",subgoal,"for",act1)
91             new_agenda = node.agenda[1:]
92             for act0 in node.actions:
93                 if (self.achieves(act0, subgoal) and
94                    self.possible((act0,act1),node.constraints)):
95                     self.display(2," reusing",act0)
96                     consts1 =
                           self.add_constraint((act0,act1),node.constraints)
97                     new_clink = (act0,subgoal,act1)
98                     new_cls = node.causal_links + [new_clink]
99                     for consts2 in
                           self.protect_cl_for_actions(node.actions,consts1,new_clink):
100                        yield Arc(node,
```

```
                                POP_node(node.actions,consts2,new_agenda,new_cls),
                                cost=0)
            for a0 in self.planning_problem.prob_domain.actions: #a0 is an
                action
                if self.achieves(a0, subgoal):
                    #a0 achieves subgoal
                    new_a = Action_instance(a0)
                    self.display(2," using new action",new_a)
                    new_actions = node.actions + [new_a]
                    consts1 =
                        self.add_constraint((self.start,new_a),node.constraints)
                    consts2 = self.add_constraint((new_a,act1),consts1)
                    new_agenda1 = new_agenda + [(pre,new_a) for pre in
                        a0.preconds.items()]
                    new_clink = (new_a,subgoal,act1)
                    new_cls = node.causal_links + [new_clink]
                    for consts3 in
                        self.protect_all_cls(node.causal_links,new_a,consts2):
                        for consts4 in
                            self.protect_cl_for_actions(node.actions,consts3,new_clink):
                            yield Arc(node,
                                      POP_node(new_actions,consts4,new_agenda1,new_cls),
                                      cost=1)
```

Given a causal link $(a0, subgoal, a1)$, the following method protects the causal link from each action in *actions*. Whenever an action deletes *subgoal*, the action needs to be before $a0$ or after $a1$. This method enumerates all constraints that result from protecting the causal link from all actions.

stripsPOP.py — (continued)

```
    def protect_cl_for_actions(self, actions, constrs, clink):
        """yields constraints that extend constrs and
        protect causal link (a0, subgoal, a1)
        for each action in actions
        """
        if actions:
            a = actions[0]
            rem_actions = actions[1:]
            a0, subgoal, a1 = clink
            if a != a0 and a != a1 and self.deletes(a,subgoal):
                if self.possible((a,a0),constrs):
                    new_const = self.add_constraint((a,a0),constrs)
                    for e in
                        self.protect_cl_for_actions(rem_actions,new_const,clink):
                        yield e # could be "yield from"
                if self.possible((a1,a),constrs):
                    new_const = self.add_constraint((a1,a),constrs)
                    for e in
                        self.protect_cl_for_actions(rem_actions,new_const,clink):
                        yield e
            else:
```

```
137             for e in
                    self.protect_cl_for_actions(rem_actions,constrs,clink):
                    yield e
138         else:
139             yield constrs
```

Given an action *act,* the following method protects all the causal links in *clinks* from *act*. Whenever *act* deletes *subgoal* from some causal link $(a0, subgoal, a1)$, the action *act* needs to be before $a0$ or after $a1$. This method enumerates all constraints that result from protecting the causal links from *act*.

stripsPOP.py — (continued)

```
141     def protect_all_cls(self, clinks, act, constrs):
142         """yields constraints that protect all causal links from act"""
143         if clinks:
144             (a0,cond,a1) = clinks[0] # select a causal link
145             rem_clinks = clinks[1:] # remaining causal links
146             if act != a0 and act != a1 and self.deletes(act,cond):
147                 if self.possible((act,a0),constrs):
148                     new_const = self.add_constraint((act,a0),constrs)
149                     for e in self.protect_all_cls(rem_clinks,act,new_const):
                            yield e
150                 if self.possible((a1,act),constrs):
151                     new_const = self.add_constraint((a1,act),constrs)
152                     for e in self.protect_all_cls(rem_clinks,act,new_const):
                            yield e
153             else:
154                 for e in self.protect_all_cls(rem_clinks,act,constrs): yield
                        e
155         else:
156             yield constrs
```

The following methods check whether an action (or action instance) achieves or deletes some subgoal.

stripsPOP.py — (continued)

```
158     def achieves(self,action,subgoal):
159         var,val = subgoal
160         return var in self.effects(action) and self.effects(action)[var] ==
                val
161
162     def deletes(self,action,subgoal):
163         var,val = subgoal
164         return var in self.effects(action) and self.effects(action)[var] !=
                val
165
166     def effects(self,action):
167         """returns the variable:value dictionary of the effects of action.
168         works for both actions and action instances"""
169         if isinstance(action, Action_instance):
170             action = action.action
```

```
        if action == "start":
            return self.planning_problem.initial_state
        elif action == "finish":
            return {}
        else:
            return action.effects
```

The constraints are represented as a set of pairs closed under transitivity. Thus if (a, b) and (b, c) are the list, then (a, c) must also be in the list. This means that adding a new constraint means adding the implied pairs, but querying whether some order is consistent is quick.

stripsPOP.py — (continued)

```
    def add_constraint(self, pair, const):
        if pair in const:
            return const
        todo = [pair]
        newconst = const.copy()
        while todo:
            x0,x1 = todo.pop()
            newconst.add((x0,x1))
            for x,y in newconst:
                if x==x1 and (x0,y) not in newconst:
                    todo.append((x0,y))
                if y==x0 and (x,x1) not in newconst:
                    todo.append((x,x1))
        return newconst

    def possible(self,pair,constraint):
        (x,y) = pair
        return (y,x) not in constraint
```

Some code for testing:

stripsPOP.py — (continued)

```
from searchBranchAndBound import DF_branch_and_bound
from searchMPP import SearcherMPP
import stripsProblem

rplanning0 = POP_search_from_STRIPS(stripsProblem.problem0)
rplanning1 = POP_search_from_STRIPS(stripsProblem.problem1)
rplanning2 = POP_search_from_STRIPS(stripsProblem.problem2)
searcher0 = DF_branch_and_bound(rplanning0,5)
searcher0a = SearcherMPP(rplanning0)
searcher1 = DF_branch_and_bound(rplanning1,10)
searcher1a = SearcherMPP(rplanning1)
searcher2 = DF_branch_and_bound(rplanning2,10)
searcher2a = SearcherMPP(rplanning2)
# Try one of the following searchers
# a = searcher0.search()
# a = searcher0a.search()
```

```
# a.end().extract_plan() # print a plan found
# a.end().constraints  # print the constraints
# SearcherMPP.max_display_level = 0 # less detailed display
# DF_branch_and_bound.max_display_level = 0 # less detailed display
# a = searcher1.search()
# a = searcher1a.search()
# a = searcher2.search()
# a = searcher2a.search()
```

Chapter 7

Supervised Machine Learning

This first chapter on machine learning covers the following topics:

- Data: how to load it, splitting into training, validation and test sets
- Features: many of the features come directly from the data. Sometimes it is useful to construct features, e.g. $height > 1.9m$ might be a Boolean feature constructed from the real-values feature *height*. The next chapter is about neural networks and how to learn features; the code in this chapter constructs them explicitly in what is often known as **feature engineering**.
- Learning with no input features: this is the base case of many methods. What should you predict if you have no input features? This provides the base cases for many algorithms (e.g., decision tree algorithm) and baselines that more sophisticated algorithms need to beat. It also provides ways to test various predictors.
- Decision tree learning: one of the classic and simplest learning algorithms, which is the basis of many other algorithms.
- Cross validation and parameter tuning: methods to prevent overfitting.
- Linear regression and classification: other classic and simple techniques that often work well (particularly combined with feature learning or engineering).
- Boosting: combining simpler learning methods to make even better learners.

A good source of classic datasets is the UCI Machine Learning Repository `https://archive.ics.uci.edu/datasets` [Lichman, 2013] [Dua and Graff, 2017]. The SPECT, IRIS, and car datasets (carbool is a Boolean version of the car dataset) are from this repository.

Dataset	# Examples	#Columns	Input Types	Target Type
SPECT	267	23	Boolean	Boolean
IRIS	150	5	numeric	categorical
car	1728	7	categorical/numeric	categorical
carbool	1728	7	categorical/numeric	Boolean
holiday	32	6	Boolean	Boolean
mail_reading	28	5	Boolean	Boolean
tv_likes	12	5	Boolean	Boolean
simp_regr	7	2	numeric	numeric

Figure 7.1: Some of the datasets used here.

7.1 Representations of Data and Predictions

The code uses the following definitions and conventions:

- A **dataset** is an enumeration of examples.
- An **example** is a list (or tuple) of values. The values can be numbers or strings.
- A **feature** is a function from examples into the range of the feature. Each feature `f` also has the following attributes:

  `f.ftype`, the type of `f`, one of: `"boolean"`, `"categorical"`, `"numeric"`

  `f.frange`, the set of values of `f` seen in the dataset, represented as a list. The `ftype` is inferred from the `frange` if not given explicitly.

  `f.__doc__`, the docstring, a string description of `f` (for printing).

  Thus for example, a **Boolean feature** is a function from the examples into $\{False, True\}$. So, if f is a Boolean feature, $f.frange == [False, True]$, and if e is an example, $f(e)$ is either *True* or *False*.

learnProblem.py — A Learning Problem

```
import math, random, statistics
import csv
from display import Displayable
from utilities import argmax

boolean = [False, True]
```

A dataset is partitioned into a training set (`train`), a validation set (`valid`) and a test set (`test`). The target feature is the feature that a learner making a prediction of. A dataset `ds` has the following attributes:

`ds.train` a list of the training examples

`ds.valid` a list of the validation examples

`ds.test` a list of the test examples

`ds.target_index` the index of the target

`ds.target` the feature corresponding to the target (a function from examples to target value)

`ds.input_features` a list of the input features

learnProblem.py — (continued)

```
18 class Data_set(Displayable):
19     """ A dataset consists of a list of training data and a list of test
           data.
20     """
21
22     def __init__(self, train, test=None, target_index=0, prob_test=0.10,
23                  prob_valid=0.11, header=None, target_type= None,
24                  one_hot=False, seed=None):
25         """A dataset for learning.
26         train is a list of tuples representing the training examples
27         test is the list of tuples representing the test examples
28         if test is None, a test set is created by selecting each
29             example with probability prob_test
30         target_index is the index of the target.
31             If negative, it counts from right.
32             If target_index is larger than the number of properties,
33             there is no target (for unsupervised learning)
34         prob_valid proability a non-test example is in validation set
35         header is a list of names for the features
36         target_type is either None for automatic detection of target type
37              or one of "numeric", "boolean", "categorical"
38         one_hot is True gives a one-hot encoding of categorical features
39         seed is for random number; None gives a different test set each time
40         """
41         if seed: # given seed makes partition consistent from run-to-run
42             random.seed(seed)
43         if test is None:
44             train,test = partition_data(train, prob_test)
45         self.train, self.valid = partition_data(train, prob_valid)
46         self.test = test
47
48         self.display(1,"Training set has",len(self.train),"examples. Number
               of columns: ",{len(e) for e in self.train})
49         self.display(1,"Test set has",len(test),"examples. Number of
               columns: ",{len(e) for e in test})
50         self.display(1,"Validation set has",len(self.valid),"examples.
               Number of columns: ",{len(e) for e in self.valid})
51         self.prob_test = prob_test
52         self.num_properties = len(self.train[0])
53         if target_index < 0: #allows for -1, -2, etc.
```

```
            self.target_index = self.num_properties + target_index
        else:
            self.target_index = target_index
        self.header = header
        self.domains = [set() for i in range(self.num_properties)]
        for example in self.train:
            for ind,val in enumerate(example):
                self.domains[ind].add(val)
        self.conditions_cache = {} # cache for computed conditions
        self.create_features(one_hot)
        if target_type:
            self.target.ftype = target_type
        self.display(1,"There are",len(self.input_features),"input
            features")

    def __str__(self):
        if self.train and len(self.train)>0: # has training examples
            return (f"Data: {len(self.train)} training, {len(self.valid)}
                validation"
                    "{len(self.test)} test examples; {len(self.train[0])}
                        features.")
        else:
            return (f"Data: {len(self.train)} training, {len(self.valid)}
                validation"
                    "{len(self.test)} test examples")
```

A **feature** is a function that takes an example and returns a value in the range of the feature. Each feature has a **frange**, which gives the range of the feature, and an **ftype** that gives the type, one of "boolean", "numeric" or "categorical".

learnProblem.py — (continued)

```
    def create_features(self, one_hot=False):
        """create the set of features.
        if one_hot==True makes categorical input features into Booleans
        """
        self.target = None
        self.input_features = []
        for i in range(self.num_properties):
            frange = list(self.domains[i])
            ftype = self.infer_type(frange)
            if one_hot and ftype == "categorical" and i !=
                self.target_index:
                for val in frange:
                    def feat(e,index=i,val=val):
                        return e[index]==val
                    if self.header:
                        feat.__doc__ = self.header[i]+"="+val
                    else:
                        feat.__doc__ = f"e[{i}]={val}"
                    feat.frange = boolean
```

```
94                  feat.type = "boolean"
95                  self.input_features.append(feat)
96              else:
97                  def feat(e,index=i):
98                    return e[index]
99                  if self.header:
100                     feat.__doc__ = self.header[i]
101                 else:
102                     feat.__doc__ = "e["+str(i)+"]"
103                 feat.frange = frange
104                 feat.ftype = ftype
105                 if i == self.target_index:
106                     self.target = feat
107                 else:
108                     self.input_features.append(feat)
```

The following tries to infer the type of each feature. Sometimes this can be wrong, (e.g., when the numbers are really categorical) and may need to be set explicitly.

learnProblem.py — (continued)

```
110     def infer_type(self,domain):
111         """Infers the type of a feature with domain
112         """
113         if all(v in {True,False} for v in domain) or all(v in {0,1} for v
                in domain):
114             return "boolean"
115         if all(isinstance(v,(float,int)) for v in domain):
116             return "numeric"
117         else:
118             return "categorical"
```

7.1.1 Creating Boolean Conditions from Features

Some of the algorithms require Boolean input features (features with range $\{0,1\}$). In order to be able to use these algorithms on datasets with arbitrary domains of input variables, the following code constructs Boolean conditions from the attributes.

There are 3 cases:

- When the range only has two values, one is designated to be the "true" value.
- When the values are all numeric, assume they are ordered (as opposed to just being some classes that happen to be labelled with numbers) and construct Boolean features for splits of the data. That is, the feature is $e[ind] < cut$ for some value cut. The number of cut values is less than or equal to `max_num_cuts`.

- When the values are not all numeric, it creates an indicator function for each value. An indicator function for a value returns true when that value is given and false otherwise. Note that we can't create an indicator function for values that appear in the test set but not in the training or validation sets because we haven't seen the test set. For the examples in the test set with a value that doesn't appear in the training set for that feature, the indicator functions all return false.

There is also an option `categorical_only` to create only Boolean features for categorical input features, and not to make cuts for numerical values.

learnProblem.py — (continued)

```
    def conditions(self, max_num_cuts=8, categorical_only = False):
        """returns a list of boolean conditions from the input features
        max_num_cuts: maximum number of cute for numeric features
        categorical_only: only categorical features are made binary
        """
        if (max_num_cuts, categorical_only) in self.conditions_cache:
            return self.conditions_cache[(max_num_cuts, categorical_only)]
        conds = []
        for ind,frange in enumerate(self.domains):
            if ind != self.target_index and len(frange)>1:
                if len(frange) == 2:
                    # two values, the feature is equality to one of them.
                    true_val = list(frange)[1] # choose one as true
                    def feat(e, i=ind, tv=true_val):
                        return e[i]==tv
                    if self.header:
                        feat.__doc__ = f"{self.header[ind]}=={true_val}"
                    else:
                        feat.__doc__ = f"e[{ind}]=={true_val}"
                    feat.frange = boolean
                    feat.ftype = "boolean"
                    conds.append(feat)
                elif all(isinstance(val,(int,float)) for val in frange):
                    if categorical_only: # numeric, don't make cuts
                        def feat(e, i=ind):
                            return e[i]
                        feat.__doc__ = f"e[{ind}]"
                        conds.append(feat)
                    else:
                        # all numeric, create cuts of the data
                        sorted_frange = sorted(frange)
                        num_cuts = min(max_num_cuts,len(frange))
                        cut_positions = [len(frange)*i//num_cuts for i in
                            range(1,num_cuts)]
                        for cut in cut_positions:
                            cutat = sorted_frange[cut]
                            def feat(e, ind_=ind, cutat=cutat):
                                return e[ind_] < cutat
```

```
                        if self.header:
                            feat.__doc__ = self.header[ind]+"<"+str(cutat)
                        else:
                            feat.__doc__ = "e["+str(ind)+"]<"+str(cutat)
                        feat.frange = boolean
                        feat.ftype = "boolean"
                        conds.append(feat)
                else:
                    # create an indicator function for every value
                    for val in frange:
                        def feat(e, ind_=ind, val_=val):
                            return e[ind_] == val_
                        if self.header:
                            feat.__doc__ = self.header[ind]+"=="+str(val)
                        else:
                            feat.__doc__= "e["+str(ind)+"]=="+str(val)
                        feat.frange = boolean
                        feat.ftype = "boolean"
                        conds.append(feat)
        self.conditions_cache[(max_num_cuts, categorical_only)] = conds
        return conds
```

Exercise 7.1 Change the code so that it splits using $e[ind] \leq cut$ instead of $e[ind] < cut$. Check boundary cases, such as 3 elements with 2 cuts. As a test case, make sure that when the range is the 30 integers from 100 to 129, and you want 2 cuts, the resulting Boolean features should be $e[ind] \leq 109$ and $e[ind] \leq 119$ to make sure that each of the resulting domains is of equal size.

Exercise 7.2 This splits on whether the feature is less than one of the values in the training set. Sam suggested it might be better to split between the values in the training set, and suggested using

$$cutat = (sorted\_frange[cut] + sorted\_frange[cut-1])/2$$

Why might Sam have suggested this? Does this work better? (Try it on a few datasets).

7.1.2 Evaluating Predictions

A **predictor** is a function that takes an example and makes a prediction on the values of the target features.

A **loss** takes a prediction and the actual value and returns a non-negative real number; lower is better. The **error** for a dataset is either the mean loss, or sometimes the sum of the losses; they differ by a constant (the number of examples). When reporting results the mean is usually used, as it can be interpreted indepoendently of the dataset size. When it is the sum, this will be made explicit.

The function `evaluate_dataset` returns the average error for each example, where the error for each example depends on the evaluation criteria.

learnProblem.py — (continued)

```
    def evaluate_dataset(self, data, predictor, error_measure):
        """Evaluates predictor on data according to the error_measure
        predictor is a function that takes an example and returns a
               prediction for the target features.
        error_measure(prediction,actual) -> non-negative real
        """
        if data:
            try:
                value = statistics.mean(error_measure(predictor(e),
                    self.target(e))
                           for e in data)
            except ValueError: # if error_measure gives an error
                return float("inf") # infinity
            return value
        else:
            return math.nan # not a number
```

Three losses are implemented: the squared or L2 loss (average of the square of the difference between the actual and predicted values), absolute or L1 loss (average of the absolute difference between the actual and predicted values) and the log loss (the average negative log-likelihood, which can be interpreted as the number of bits to describe an example using a code based on the prediction treated as a probability). The accuracy is also defined, but it is not a loss as it should be maximized.

This is defined using a class, `Evaluate` but no instances will be created. Just use `Evaluate.squared_loss` etc. (Please keep the __doc__ strings a consistent length as they are used in tables.) The prediction is either a real value or a $\{value : probability\}$ dictionary or a list. The actual is either a real number or a key of the prediction.

learnProblem.py — (continued)

```
class Evaluate(object):
    """A container for the evaluation measures"""

    def squared_loss(prediction, actual):
        "squared loss "
        if isinstance(prediction, (list,dict)):
            return (1-prediction[actual])**2 # the correct value is 1
        else:
            return (prediction-actual)**2

    def absolute_loss(prediction, actual):
        "absolute loss "
        if isinstance(prediction, (list,dict)):
            return abs(1-prediction[actual]) # the correct value is 1
        else:
            return abs(prediction-actual)

    def log_loss(prediction, actual):
```

```
214         "log loss (bits)"
215         try:
216             if isinstance(prediction, (list,dict)):
217                 return -math.log2(prediction[actual])
218             else:
219                 return -math.log2(prediction) if actual==1 else
                        -math.log2(1-prediction)
220         except ValueError:
221             return float("inf") # infinity
222
223     def accuracy(prediction, actual):
224         "accuracy    "
225         return themode(prediction) == actual
226
227     all_criteria = [accuracy, absolute_loss, squared_loss, log_loss]
228
229 def themode(prediction):
230     """the mode of a prediction.
231     This handles all of the cases of AIPython predictors: dictionaries,
            lists and boolean probabilities.
232     """
233     if isinstance(prediction, dict):
234         md, val = None, -math.inf
235         for (p,v) in prediction.items():
236             if v> val:
237                 md, val = p,v
238         return md
239     if isinstance(prediction, list):
240         md,val = 0,prediction[0]
241         for i in range(1,len(prediction)):
242             if prediction[i]>val:
243                 md,val = i,prediction[i]
244         return md
245     else: # prediction is probability of Boolean
246         return False if prediction < 0.5 else True
```

7.1.3 Creating Test and Training Sets

The following method partitions the data into a training set and a test set. (Also training into training and validation sets). Note that this does not guarantee that the test set will contain exactly a proportion of the data equal to `prob_test`.

[An alternative is to use `random.sample()` which can guarantee that the test set will contain exactly a particular proportion of the data. However this would require knowing how many elements are in the dataset, which it may not know, as `data` may just be a generator of the data (e.g., when reading the data from a file).]

learnProblem.py — (continued)

```
248 def partition_data(data, prob_test=0.30):
```

```
249     """partitions the data into a training set and a test set, where
250     prob_test is the probability of each example being in the test set.
251     """
252     train = []
253     test = []
254     for example in data:
255         if random.random() < prob_test:
256             test.append(example)
257         else:
258             train.append(example)
259     return train, test
```

7.1.4 Importing Data From File

A dataset is typically loaded from a file. The default here is that it loaded from a CSV (comma separated values) file, although the separator can be changed. This assumes that all lines that contain the separator are valid data (so it only includes those data items that contain more than one element). This allows for blank lines and comment lines that do not contain the separator. However, it means that this method is not suitable for cases where there is only one feature.

Note that *data_all* and *data_tuples* are generators. *data_all* is a generator of a list of list of strings. This version assumes that CSV files are simple. The standard `csv` package, that allows quoted arguments, can be used by uncommenting the line for *data_all* and commenting out the line that follows. *data_tuples* contains only those lines that contain the delimiter (others lines are assumed to be empty or comments), and tries to convert the elements to numbers whenever possible.

learnProblem.py — (continued)

```
261 class Data_from_file(Data_set):
262     def __init__(self, file_name, separator=',', num_train=None,
        prob_test=0.10, prob_valid=0.11,
263             has_header=False, target_index=0, one_hot=False,
264             categorical=[], target_type= None, seed=None):
265         """create a dataset from a file
266         separator is the character that separates the attributes (',' for
            CSV file)
267         num_train is a number specifying the first num_train tuples are
            training, or None
268         prob_test is the probability each example is in the test set (if
            num_train is None)
269         prob_valid is the probability each non-test example is in the
            validation set
270         has_header is True if the first line of file is a header
271         target_index specifies which feature is the target
272         one_hot specifies whether categorical features should be encoded as
            one_hot.
```

```
        categorical is a set (or list) of features that should be treated as categorical
        target_type is either None for automatic detection of target type or one of "numeric", "boolean", "categorical"
        """
        with open(file_name,'r',newline='') as csvfile:
            self.display(1,"Loading",file_name)
            # data_all = csv.reader(csvfile,delimiter=separator) # for more complicated CSV files
            data_all = (line.strip().split(separator) for line in csvfile)
            if has_header:
                header = next(data_all)
            else:
                header = None
            data_tuples = (interpret_elements(d) for d in data_all if len(d)>1)
            if num_train is not None:
                # training set is divided into training then text examples
                # the file is only read once, and the data is placed in appropriate list
                train = []
                for i in range(num_train):  # will give an error if insufficient examples
                    train.append(next(data_tuples))
                test = list(data_tuples)
                Data_set.__init__(self,train, test=test,
                    prob_valid=prob_valid,
                                target_index=target_index,header=header,
                                    seed=seed,
                                target_type=target_type, one_hot=one_hot)
            else:    # randomly assign training and test examples
                Data_set.__init__(self,data_tuples, test=None,
                    prob_test=prob_test, prob_valid=prob_valid,
                                target_index=target_index, header=header,
                                    seed=seed,
                                target_type=target_type, one_hot=one_hot)
```

The following class is used for datasets where the training and test are in different files

learnProblem.py — (continued)

```
class Data_from_files(Data_set):
    def __init__(self, train_file_name, test_file_name, separator=',',
                 has_header=False, target_index=0, one_hot=False,
                 categorical=[], target_type= None):
        """create a dataset from separate training and file
        separator is the character that separates the attributes
        num_train is a number specifying the first num_train tuples are training, or None
        prob_test is the probability an example should in the test set (if num_train is None)
```

```
        has_header is True if the first line of file is a header
        target_index specifies which feature is the target
        one_hot specifies whether categorical features should be encoded as
            one-hot
        categorical is a set (or list) of features that should be treated
            as categorical
        target_type is either None for automatic detection of target type
             or one of "numeric", "boolean", "categorical"
        """
        with open(train_file_name,'r',newline='') as train_file:
          with open(test_file_name,'r',newline='') as test_file:
            # data_all = csv.reader(csvfile,delimiter=separator) # for more
                complicated CSV files
            train_data = (line.strip().split(separator) for line in
                train_file)
            test_data = (line.strip().split(separator) for line in
                test_file)
            if has_header: # this assumes the training file has a header
                and the test file doesn't
                header = next(train_data)
            else:
                header = None
            train_tuples = [interpret_elements(d) for d in train_data if
                len(d)>1]
            test_tuples = [interpret_elements(d) for d in test_data if
                len(d)>1]
            Data_set.__init__(self,train_tuples, test_tuples,
                              target_index=target_index, header=header,
                                  one_hot=one_hot)
```

When reading from a file all of the values are strings. This next method tries to convert each value into a number (an int or a float) or Boolean, if it is possible.

learnProblem.py — (continued)

```
def interpret_elements(str_list):
    """make the elements of string list str_list numeric if possible.
    Otherwise remove initial and trailing spaces.
    """
    res = []
    for e in str_list:
        try:
            res.append(int(e))
        except ValueError:
            try:
                res.append(float(e))
            except ValueError:
                se = e.strip()
                if se in ["True","true","TRUE"]:
                    res.append(True)
                elif se in ["False","false","FALSE"]:
```

```
346                 res.append(False)
347             else:
348                 res.append(e.strip())
349     return res
```

7.1.5 Augmented Features

Sometimes we want to augment the features with new features computed from the old features (e.g., the product of features). The following code creates a new dataset from an old dataset but with new features. Note that special cases of these are **kernel**s; mapping the original feature space into a new space, which allow a neat way to do learning in the augmented space for many mappings (the "kernel trick"). This is beyond the scope of AIPython; those interested should read about *support vector machines*.

Reacall that a feature is a function of examples. A unary feature constructor takes a feature and returns a new feature. A binary feature combiner takes two features and returns a new feature.

learnProblem.py — (continued)

```
351 class Data_set_augmented(Data_set):
352     def __init__(self, dataset, unary_functions=[], binary_functions=[],
            include_orig=True):
353         """creates a dataset like dataset but with new features
354         unary_function is a list of unary feature constructors
355         binary_functions is a list of binary feature combiners.
356         include_orig specifies whether the original features should be
                included
357         """
358         self.orig_dataset = dataset
359         self.unary_functions = unary_functions
360         self.binary_functions = binary_functions
361         self.include_orig = include_orig
362         self.target = dataset.target
363         Data_set.__init__(self,dataset.train, test=dataset.test,
364                           target_index = dataset.target_index)
365
366     def create_features(self, one_hot=False):
367         """create the set of features.
368            one_hot is ignored, but could be implemented as in
                   Data_set.create_features
369         """
370         if self.include_orig:
371             self.input_features = self.orig_dataset.input_features.copy()
372         else:
373             self.input_features = []
374         for u in self.unary_functions:
375             for f in self.orig_dataset.input_features:
376                 self.input_features.append(u(f))
377         for b in self.binary_functions:
```

```
            for f1 in self.orig_dataset.input_features:
                for f2 in self.orig_dataset.input_features:
                    if f1 != f2:
                        self.input_features.append(b(f1,f2))
```

The following are useful unary feature constructors and binary feature combiner.

learnProblem.py — (continued)

```
def square(f):
    """a unary feature constructor to construct the square of a feature
    """
    def sq(e):
        return f(e)**2
    sq.__doc__ = f.__doc__+"**2"
    return sq

def power_feat(n):
    """given n returns a unary feature constructor to construct the nth
        power of a feature.
    e.g., power_feat(2) is the same as square, defined above
    """
    def fn(f,n=n):
        def pow(e,n=n):
            return f(e)**n
        pow.__doc__ = f.__doc__+"**"+str(n)
        return pow
    return fn

def prod_feat(f1,f2):
    """a new feature that is the product of features f1 and f2
    """
    def feat(e):
        return f1(e)*f2(e)
    feat.__doc__ = f1.__doc__+"*"+f2.__doc__
    return feat

def eq_feat(f1,f2):
    """a new feature that is 1 if f1 and f2 give same value
    """
    def feat(e):
        return 1 if f1(e)==f2(e) else 0
    feat.__doc__ = f1.__doc__+"=="+f2.__doc__
    return feat

def neq_feat(f1,f2):
    """a new feature that is 1 if f1 and f2 give different values
    """
    def feat(e):
        return 1 if f1(e)!=f2(e) else 0
    feat.__doc__ = f1.__doc__+"!="+f2.__doc__
```

```
424     return feat
```

Example:

learnProblem.py — (continued)

```
426 # from learnProblem import Data_set_augmented,prod_feat
427 # data = Data_from_file('data/holiday.csv', has_header=True, num_train=19,
        target_index=-1)
428 # data = Data_from_file('data/iris.data', prob_test=1/3, target_index=-1)
429 ## Data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0)
430 # dataplus = Data_set_augmented(data,[],[prod_feat])
431 # dataplus = Data_set_augmented(data,[],[prod_feat,neq_feat])
```

Exercise 7.3 For symmetric properties, such as product, we don't need both $f1 * f2$ as well as $f2 * f1$ as extra properties. Allow the user to be able to declare feature constructors as symmetric (by associating a Boolean feature with them). Change *construct_features* so that it does not create both versions for symmetric combiners.

7.2 Generic Learner Interface

A **learner** takes a dataset (and possibly other arguments specific to the method). To get it to learn, call the *learn*() method. This implements *Displayable* so that it can display traces at multiple levels of detail (perhaps with a GUI).

learnProblem.py — (continued)

```
432 from display import Displayable
433
434 class Learner(Displayable):
435     def __init__(self, dataset):
436         raise NotImplementedError("Learner.__init__") # abstract method
437
438     def learn(self):
439         """returns a predictor, a function from a tuple to a value for the
                target feature
440         """
441         raise NotImplementedError("learn") # abstract method
442
443     def __str__(self, sig_dig=3):
444         """String reprenentation of the learned predictor
445         """
446         return "no representation"
447
448     def evaluate(self):
449         """Tests default learner on data
450         """
451         self.learn()
452         print(f"function learned is {self}")
453         print("Criterion\tTraining\tvalidation\ttest")
```

```
454         for ecrit in Evaluate.all_criteria:
455             print(ecrit.__doc__, end='\t')
456             for data_subset in [self.dataset.train, self.dataset.valid,
                    self.dataset.test]:
457                 error = self.dataset.evaluate_dataset(data_subset,
                        self.predictor, ecrit)
458                 print(str(round(error,7)), end='\t')
459             print()
```

7.3 Learning With No Input Features

If you need make the same prediction for each example (the input features are ignored), what prediction should you make? This can be used as a naive baseline; if a more sophisticated method does not do better than this, it is not useful. This also provides the base case for some methods, such as decision-tree learning.

> To run demo to compare different prediction methods on various evaluation criteria, in folder "aipython", load "learnNoInputs.py", using e.g., ipython -i learnNoInputs.py, and it prints some test results.

There are a few alternatives as to what could be allowed in a prediction:

- a point prediction, where only allowed the values of the feature can predicted. For example, if the values of the feature are $\{0,1\}$ we are only allowed to predict 0 or 1 or of the values are ratings in $\{1,2,3,4,5\}$, we can only predict one of these integers.

- a point prediction, where any value can be predicted. For example, if the values of the feature are $\{0,1\}$ it could predict 0.3, 1, or even 1.7. For all of the criteria defined, there is no point in predicting a value greater than 1 or less that zero (but that doesn't mean it can't). If the values are ratings in $\{1,2,3,4,5\}$, you may want to predict 3.4.

- a probability distribution over the values of the feature. For each value v, it predicts a non-negative number p_v, such that the sum over all predictions is 1.

Here are some prediction functions that take in an enumeration of values, a domain, and returns a point prediction: a value or dictionary of $\{value : prediction\}$. Note that cmedian returns one of the middle values when there are an even number of examples, whereas median gives the average of them (and so cmedian is applicable for ordinals that cannot be considered cardinal values). Similarly, cmode picks one of the values when more than one value has the maximum number of elements.

learnNoInputs.py — Learning ignoring all input features

```
from learnProblem import Evaluate
import math, random, collections, statistics
import utilities # argmax for (element,value) pairs

class Predict(object):
    """The class of prediction methods for a list of values.
    The doc strings the same length because they are used in tables.
    Note that the methods don't have the self argument.
    To use call Predict.laplace(data) etc."""

    ### The following return a distribution over values (for classification)
    def empirical(data, domain=[0,1], icount=0):
        "empirical dist "
        # returns a distribution over values
        # icount is pseudo count for each value
        counts = {v:icount for v in domain}
        for e in data:
            counts[e] += 1
        s = sum(counts.values())
        return {k:v/s for (k,v) in counts.items()}

    def laplace(data, domain=[0,1]):
        "Laplace       "  # for categorical data
        return Predict.empirical(data, domain, icount=1)

    def cmode(data, domain=[0,1]):
        "mode          " # for categorical data
        md = statistics.mode(data)
        return {v: 1 if v==md else 0 for v in domain}

    def cmedian(data, domain=[0,1]):
        "median        " # for categorical data
        md = statistics.median_low(data) # always return one of the values
        return {v: 1 if v==md else 0 for v in domain}

    ### The following return a single prediction (for regression).
    ### The domain argument is ignored.

    def mean(data, domain=[0,1]):
        "mean          "
        # returns a real number
        return statistics.mean(data)

    def rmean(data, domain=[0,1], mean0=0, pseudo_count=1):
        "regularized mean"
        # returns a real number.
        # mean0 is the mean to be used for 0 data points
        # With mean0=0.5, pseudo_count=2, same as laplace for [0,1] data
        sm = mean0 * pseudo_count
```

```
60          count = pseudo_count
61          for e in data:
62              sm += e
63              count += 1
64          return sm/count
65
66      def mode(data, domain=[0,1]):
67          "mode          "
68          return statistics.mode(data)
69
70      def median(data, domain=[0,1]):
71          "median        "
72          return statistics.median(data)
73
74      all = [empirical, mean, rmean, laplace, cmode, mode, median,cmedian]
75
76      # The following suggests appropriate predictions as a function of the
            target type
77      select = {"boolean": [empirical, laplace, cmode, cmedian],
78                "categorical": [empirical, laplace, cmode, cmedian],
79                "numeric": [mean, rmean, mode, median]}
```

Exercise 7.4 Create a predictor `bounded_empirical` which is like `empirical` but avoids predictions of 0 or 1 (which can give errors for log loss), by using using some ϵ instead of 0 and $1-\epsilon$ instead of 1, and otherise uses the empirical mean.

7.3.1 Evaluation

To evaluate a point prediction, let's first generate some possible values, 0 and 1 for the target feature. Given the ground truth *prob*, a number in the range $[0,1]$, the following code generates some training and test data where *prob* is the probability of each example being 1. To generate a 1 with probability *prob*, it generates a random number in range [0,1] and return 1 if that number is less than *prob*. A prediction is computed by applying the predictor to the training data, which is evaluated on the test set. This is repeated `num_samples` times.

Let's evaluate the predictions of the possible selections according to the different evaluation criteria, for various training sizes.

learnNoInputs.py — (continued)

```
81  def test_no_inputs(error_measures = Evaluate.all_criteria,
        num_samples=10000,
82                     test_size=10, training_sizes=
                           [1,2,3,4,5,10,20,100,1000]):
83      for train_size in training_sizes:
84          results = {predictor: {error_measure: 0 for error_measure in
                error_measures}
85                     for predictor in Predict.all}
86          for sample in range(num_samples):
87              prob = random.random()
```

```
            training = [1 if random.random()<prob else 0 for i in
                range(train_size)]
            test = [1 if random.random()<prob else 0 for i in
                range(test_size)]
            for predictor in Predict.all:
                prediction = predictor(training)
                for error_measure in error_measures:
                    results[predictor][error_measure] += sum(
                        error_measure(prediction,actual)
                                                            for actual in
                                                                test) /
                                                                test_size
        print(f"For training size {train_size}:")
        print("  Predictor\t","\t".join(error_measure.__doc__ for
                                        error_measure in
                                            error_measures),sep="\t")
        for predictor in Predict.all:
            print(f"  {predictor.__doc__}",
                      "\t".join("{:.7f}".format(results[predictor][error_measure]/num_samples)
                                    for error_measure in
                                        error_measures),sep="\t")

if __name__ == "__main__":
    test_no_inputs()
```

Exercise 7.5 Which predictor works best for low counts when the error is

(a) Squared error

(b) Absolute error

(c) Log loss

You may need to try this a few times to make sure your answer is supported by the evidence. Does the difference from the other methods get more or less as the number of examples grow?

Exercise 7.6 Suggest other predictors that only take the training data. (E.g., `bounded_empirical` of Exercise 7.4, for some ϵ or to change the pseodo-counts of the Laplace method.)

7.4 Decision Tree Learning

To run the decision tree learning demo, in folder "`aipython`", load "`learnDT.py`", using e.g., `ipython -i learnDT.py`, and it prints some test results. To try more examples, copy and paste the commented-out commands at the bottom of that file. This requires Python 3 with `matplotlib`.

The decision tree algorithm does binary splits, and assumes that all input features are binary functions of the examples.

learnDT.py — Learning a binary decision tree

```
from learnProblem import Learner, Evaluate
from learnNoInputs import Predict
import math

class DT_learner(Learner):
    def __init__(self,
                 dataset,
                 split_to_optimize=Evaluate.log_loss, # to minimize for at each split
                 leaf_prediction=Predict.empirical, # what to use for value at leaves
                 train=None,                  # used for cross validation
                 max_num_cuts=8, # maximum number of conditions to split a numeric feature into
                 gamma=1e-7,  # minimum improvement needed to expand a node
                 min_child_weight=10):
        self.dataset = dataset
        self.target = dataset.target
        self.split_to_optimize = split_to_optimize
        self.leaf_prediction = leaf_prediction
        self.max_num_cuts = max_num_cuts
        self.gamma = gamma
        self.min_child_weight = min_child_weight
        if train is None:
            self.train = self.dataset.train
        else:
            self.train = train

    def learn(self, max_num_cuts=8):
        """learn a decision tree"""
        self.predictor =
            self.learn_tree(self.dataset.conditions(self.max_num_cuts),
            self.train)
        return self.predictor

    def __str__(self):
       """string only exists after learning"""
       return self.predictor.__doc__
```

The main recursive algorithm, takes in a set of input features and a set of training data. It first decides whether to split. If it doesn't split, it makes a point prediction, ignoring the input features.

It only splits if the best split increases the error by at least `gamma`. This implies it does not split when:

- there are no more input features
- there are fewer examples than *min_number_examples*,
- all the examples agree on the value of the target, or

- the best split puts all examples in the same partition.

If it splits, it selects the best split according to the evaluation criterion (assuming that is the only split it gets to do), and returns the condition to split on (in the variable *split*) and the corresponding partition of the examples.

learnDT.py — (continued)

```
45    def learn_tree(self, conditions, data_subset):
46        """returns a decision tree
47        conditions is a set of possible conditions
48        data_subset is a subset of the data used to build this (sub)tree
49
50        where a decision tree is a function that takes an example and
51        makes a prediction on the target feature
52        """
53        self.display(2,f"learn_tree with {len(conditions)} features and
              {len(data_subset)} examples")
54        split, partn = self.select_split(conditions, data_subset)
55        if split is None: # no split; return a point prediction
56            prediction = self.leaf_value(data_subset, self.target.frange)
57            self.display(2,f"leaf prediction for {len(data_subset)}
                  examples is {prediction}")
58            def leaf_fun(e):
59                return prediction
60            leaf_fun.__doc__ = str(prediction)
61            leaf_fun.num_leaves = 1
62            return leaf_fun
63        else:  # a split succeeded
64            false_examples, true_examples = partn
65            rem_features = [fe for fe in conditions if fe != split]
66            self.display(2,"Splitting on",split.__doc__,"with examples
                  split",
67                         len(true_examples),":",len(false_examples))
68            true_tree = self.learn_tree(rem_features,true_examples)
69            false_tree = self.learn_tree(rem_features,false_examples)
70            def fun(e):
71                if split(e):
72                    return true_tree(e)
73                else:
74                    return false_tree(e)
75            #fun = lambda e: true_tree(e) if split(e) else false_tree(e)
76            fun.__doc__ = (f"(if {split.__doc__} then {true_tree.__doc__}"
77                           f" else {false_tree.__doc__})")
78            fun.num_leaves = true_tree.num_leaves + false_tree.num_leaves
79            return fun
```

learnDT.py — (continued)

```
81    def leaf_value(self, egs, domain):
82        return self.leaf_prediction((self.target(e) for e in egs), domain)
83
```

```
    def select_split(self, conditions, data_subset):
        """finds best feature to split on.

        conditions is a non-empty list of features.
        returns feature, partition
        where feature is an input feature with the smallest error as
             judged by split_to_optimize or
             feature==None if there are no splits that improve the error
        partition is a pair (false_examples, true_examples) if feature is
            not None
        """
        best_feat = None # best feature
        # best_error = float("inf") # infinity - more than any error
        best_error = self.sum_losses(data_subset) - self.gamma
        self.display(3," no split has
            error=",best_error,"with",len(conditions),"conditions")
        best_partition = None
        for feat in conditions:
            false_examples, true_examples = partition(data_subset,feat)
            if
                min(len(false_examples),len(true_examples))>=self.min_child_weight:
                err = (self.sum_losses(false_examples)
                       + self.sum_losses(true_examples))
                self.display(3," split on",feat.__doc__,"has error=",err,
                         "splits
                             into",len(true_examples),":",len(false_examples),"gamma=",self.gamma)
                if err < best_error:
                    best_feat = feat
                    best_error=err
                    best_partition = false_examples, true_examples
        self.display(2,"best split is on",best_feat.__doc__,
                             "with err=",best_error)
        return best_feat, best_partition

    def sum_losses(self, data_subset):
        """returns sum of losses for dataset (with no more splits)
        There a single prediction for all leaves using leaf_prediction
        It is evaluated using split_to_optimize
        """
        prediction = self.leaf_value(data_subset, self.target.frange)
        error = sum(self.split_to_optimize(prediction, self.target(e))
                    for e in data_subset)
        return error

def partition(data_subset,feature):
    """partitions the data_subset by the feature"""
    true_examples = []
    false_examples = []
    for example in data_subset:
        if feature(example):
```

```
130             true_examples.append(example)
131         else:
132             false_examples.append(example)
133     return false_examples, true_examples
```

Test cases:

learnDT.py — (continued)

```
136 from learnProblem import Data_set, Data_from_file
137
138 def testDT(data, print_tree=True, selections = None, **tree_args):
139     """Prints errors and the trees for various evaluation criteria and ways
            to select leaves.
140     """
141     if selections == None: # use selections suitable for target type
142         selections = Predict.select[data.target.ftype]
143     evaluation_criteria = Evaluate.all_criteria
144     print("Split Choice","Leaf Choice\t","#leaves",'\t'.join(ecrit.__doc__
145                                                 for ecrit in
                                                        evaluation_criteria),sep="\t")
146     for crit in evaluation_criteria:
147         for leaf in selections:
148             tree = DT_learner(data, split_to_optimize=crit,
                        leaf_prediction=leaf,
149                               **tree_args).learn()
150             print(crit.__doc__, leaf.__doc__, tree.num_leaves,
151                   "\t".join("{:.7f}".format(data.evaluate_dataset(data.test,
                              tree, ecrit))
152                                 for ecrit in evaluation_criteria),sep="\t")
153             if print_tree:
154                 print(tree.__doc__)
155
156 #DT_learner.max_display_level = 4 # more detailed trace
157 if __name__ == "__main__":
158     # Choose one of the data files
159     #data=Data_from_file('data/SPECT.csv', target_index=0);
            print("SPECT.csv")
160     #data=Data_from_file('data/iris.data', target_index=-1);
            print("iris.data")
161     data = Data_from_file('data/carbool.csv', one_hot=True,
            target_index=-1, seed=123)
162     #data = Data_from_file('data/mail_reading.csv', target_index=-1);
            print("mail_reading.csv")
163     #data = Data_from_file('data/holiday.csv', has_header=True,
            num_train=19, target_index=-1); print("holiday.csv")
164     testDT(data, print_tree=False)
```

Note that different runs may provide different values as they split the training and test sets differently. So if you have a hypothesis about what works better, make sure it is true for different runs.

Exercise 7.7 The current algorithm does not have a very sophisticated stopping criterion. What is the current stopping criterion? (Hint: you need to look at both *learn_tree* and *select_split*.)

Exercise 7.8 Extend the current algorithm to include in the stopping criterion

(a) A minimum child size; don't use a split if one of the children has fewer elements that this.

(b) A depth-bound on the depth of the tree.

(c) An improvement bound such that a split is only carried out if error with the split is better than the error without the split by at least the improvement bound.

Which values for these parameters make the prediction errors on the test set the smallest? Try it on more than one dataset.

Exercise 7.9 Without any input features, it is often better to include a pseudo-count that is added to the counts from the training data. Modify the code so that it includes a pseudo-count for the predictions. When evaluating a split, including pseudo counts can make the split worse than no split. Does pruning with an improvement bound and pseudo-counts make the algorithm work better than with an improvement bound by itself?

Exercise 7.10 Some people have suggested using information gain (which is equivalent to greedy optimization of log loss) as the measure of improvement when building the tree, even in they want to have non-probabilistic predictions in the final tree. Does this work better than myopically choosing the split that is best for the evaluation criteria we will use to judge the final prediction?

7.5 k-fold Cross Validation and Parameter Tuning

To run the cross validation demo, in folder "`aipython`", load "`learnCrossValidation.py`", using e.g., `ipython -i learnCrossValidation.py`. The commented-out commands at the bottom can produce a graph like Figure 7.15. Different runs will produce different graphs, so your graph will be different the one in [Poole and Mackworth, 2023].

k-fold cross validation is more sophisticated than dividing the non-test set into a training and validation set as done above. If you are doing *k*-fold cross validation, set `prob_valid` to 0 in Data, as this does its own division into validation sets.

The above decision tree algorithm tends to overfit the data. One way to determine whether the prediction is overfitting is by cross validation. The code below implements *k*-fold cross validation, which can be used to choose the

value of parameters to best fit the training data. If we want to use parameter tuning to improve predictions on a particular dataset, we can only use the training data (and not the test data) to tune the parameter.

k-fold cross validation partitions the training set into *k* approximately equal-sized folds. For each fold, it trains on the other examples, and determine the error of the prediction on that fold. For example, if there are 10 folds, it train on 90% of the data, and tests on remaining 10% of the data. It does this 10 times, so that each example gets used as a test set once, and in the training set 9 times.

The code below creates one copy of the data, and multiple views of the data. For each fold, *fold* enumerates the examples in the fold, and *fold_complement* enumerates the examples not in the fold.

learnCrossValidation.py — Cross Validation for Parameter Tuning

```
from learnProblem import Data_set, Data_from_file, Evaluate
from learnNoInputs import Predict
from learnDT import DT_learner
import matplotlib.pyplot as plt
import random

class K_fold_dataset(object):
    def __init__(self, training_set, num_folds):
        self.data = training_set.train.copy()
        self.target = training_set.target
        self.input_features = training_set.input_features
        self.num_folds = num_folds
        self.conditions = training_set.conditions

        random.shuffle(self.data)
        self.fold_boundaries = [(len(self.data)*i)//num_folds
                                for i in range(0,num_folds+1)]

    def fold(self, fold_num):
        for i in range(self.fold_boundaries[fold_num],
                       self.fold_boundaries[fold_num+1]):
            yield self.data[i]

    def fold_complement(self, fold_num):
        for i in range(0,self.fold_boundaries[fold_num]):
            yield self.data[i]
        for i in range(self.fold_boundaries[fold_num+1],len(self.data)):
            yield self.data[i]
```

The validation error is the average error for each example, where we test on each fold, and learn on the other folds.

learnCrossValidation.py — (continued)

```
    def validation_error(self, learner, error_measure, **other_params):
        error = 0
        try:
            for i in range(self.num_folds):
```

```
            predictor = learner(self,
                train=list(self.fold_complement(i)),
                           **other_params).learn()
            error += sum( error_measure(predictor(e), self.target(e))
                        for e in self.fold(i))
        except ValueError:
            return float("inf") #infinity
        return error/len(self.data)
```

The `plot_error` method plots the average error as a function of the minimum number of examples in decision-tree search, both for the validation set and for the test set. The error on the validation set can be used to tune the parameter — choose the value of the parameter that minimizes the error. The error on the test set cannot be used to tune the parameters; if it were to be used this way it could not be used to test how well the method works on unseen examples.

learnCrossValidation.py — (continued)

```
def plot_error(data, criterion=Evaluate.squared_loss,
                 leaf_prediction=Predict.empirical,
                 num_folds=5, maxx=None, xscale='linear'):
    """Plots the error on the validation set and the test set
    with respect to settings of the minimum number of examples.
    xscale should be 'log' or 'linear'
    """
    plt.ion()
    fig, ax = plt.subplots()
    ax.set_xscale(xscale) # change between log and linear scale
    ax.set_xlabel("min_child_weight")
    ax.set_ylabel("average "+criterion.__doc__)
    folded_data = K_fold_dataset(data, num_folds)
    if maxx == None:
        maxx = len(data.train)//2+1
    verrors = [] # validation errors
    terrors = [] # test set errors
    for mcw in range(1,maxx):
        verrors.append(folded_data.validation_error(DT_learner, criterion,
                                                    leaf_prediction=leaf_prediction,
                                                    min_child_weight=mcw))
        tree = DT_learner(data, criterion, leaf_prediction=leaf_prediction,
                          min_child_weight=mcw).learn()
        terrors.append(data.evaluate_dataset(data.test,tree,criterion))
    ax.plot(range(1,maxx), verrors, ls='-',color='k',
              label="validation for "+criterion.__doc__)
    ax.plot(range(1,maxx), terrors, ls='--',color='k',
              label="test set for "+criterion.__doc__)
    ax.legend()

# The following produces variants of Figure 7.18 of Poole and Mackworth
    [2023]
```

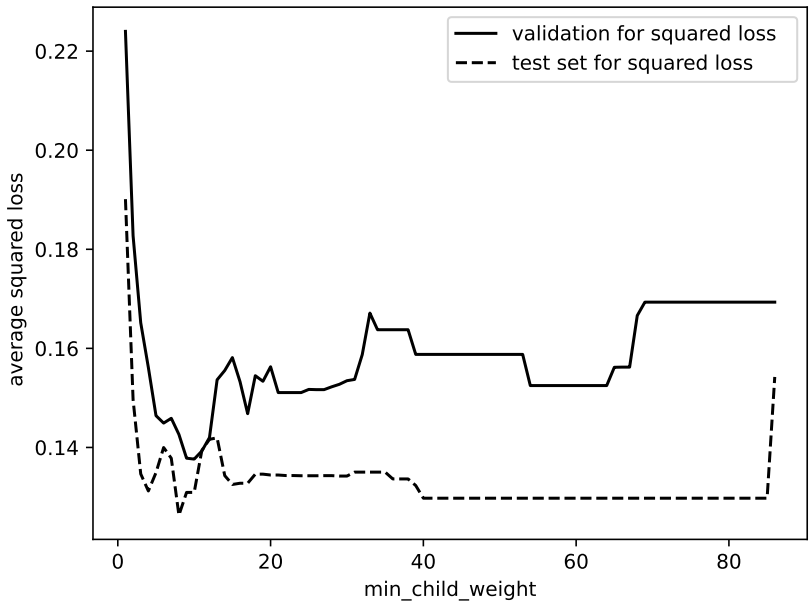


Figure 7.2: `plot_error` for SPECT dataset

```
# data = Data_from_file('data/SPECT.csv',target_index=0, prob_valid=0)
# plot_error(data, criterion=Evaluate.log_loss,
    leaf_prediction=Predict.laplace)

#alternatively try:
# plot_error(data)
# data = Data_from_file('data/carbool.csv', one_hot=True, target_index=-1,
    seed=123)
```

Figure 7.2 shows the average squared loss in the validation and test sets as a function of the `min_child_weight` in the decision-tree learning algorithmon the SPECT dataset. It was plotted with `plot_error(data)`). The assumption behind cross validation is that the parameter that minimizes the loss on the validation set, will be a good parameter for the test set.

If you rerun the `Data_from_file`, you will get the new test and training sets, and so the graph will change.

Exercise 7.11 Change the error plot so that it can evaluate the stopping criteria of the exercise of Section 7.7. Which criteria makes the most difference?

7.6 Linear Regression and Classification

Here is a stochastic gradient descent searcher for linear regression and classification.

learnLinear.py — Linear Regression and Classification

```
from learnProblem import Learner
import random, math

class Linear_learner(Learner):
    def __init__(self, dataset, train=None,
                 learning_rate=0.1, max_init = 0.2, squashed=True):
        """Creates a gradient descent searcher for a linear classifier.
        The main learning is carried out by learn()

        dataset provides the target and the input features
        train provides a subset of the training data to use
        learning_rate is the gradient descent step size
        max_init is the maximum absolute value of the initial weights
        squashed specifies whether the output is a squashed linear function
        """
        self.dataset = dataset
        self.target = dataset.target
        if train==None:
            self.train = self.dataset.train
        else:
            self.train = train
        self.learning_rate = learning_rate
        self.squashed = squashed
        self.input_features = [one]+dataset.input_features # one is defined
            below
        self.weights = {feat:random.uniform(-max_init,max_init)
                        for feat in self.input_features}
```

predictor predicts the value of an example from the current parameter settings.

learnLinear.py — (continued)

```

    def predictor(self,e):
        """returns the prediction of the learner on example e"""
        linpred = sum(w*f(e) for f,w in self.weights.items())
        if self.squashed:
            return sigmoid(linpred)
        else:
            return linpred

    def __str__(self, sig_dig=3):
        """returns the doc string for the current prediction function
        sig_dig is the number of significant digits in the numbers"""
        doc = "+".join(str(round(val,sig_dig))+"*"+feat.__doc__
```

```
                    for feat,val in self.weights.items())
        if self.squashed:
            return "sigmoid("+ doc+")"
        else:
            return doc
```

learn is the main algorithm of the learner. It does num_iter steps (batches) of stochastic gradient descent, for the given batch size.

learnLinear.py — (continued)

```
    def learn(self, batch_size=32, num_iter=100):
        batch_size = min(batch_size, len(self.train))
        d = {feat:0 for feat in self.weights}
        for it in range(num_iter):
            self.display(2,f"prediction= {self}")
            for e in random.sample(self.train, batch_size):
                error = self.predictor(e) - self.target(e)
                for feat in self.weights:
                    d[feat] += error*feat(e)
            for feat in self.weights:
                self.weights[feat] -= self.learning_rate*d[feat]
                d[feat]=0
        return self.predictor
```

one is a function that always returns 1. This is used for one of the input properties.

learnLinear.py — (continued)

```
def one(e):
    "1"
    return 1
```

$sigmoid(x)$ is the function

$$\frac{1}{1+e^{-x}}$$

The inverse of *sigmoid* is the *logit* function

learnLinear.py — (continued)

```
def sigmoid(x):
    return 1/(1+math.exp(-x))

def logit(x):
    return -math.log(1/x-1)
```

softmax$([x_0, x_2, \ldots])$ returns $[v_0, v_2, \ldots]$ where

$$v_i = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

learnLinear.py — (continued)

```
def softmax(xs, domain=None):
    """xs is a list of values, and
    domain is the domain (a list) or None if the list should be returned
    returns a distribution over the domain (a dict)
    """
    m = max(xs) # use of m prevents overflow (and all values underflowing)
    exps = [math.exp(x-m) for x in xs]
    s = sum(exps)
    if domain:
        return {d:v/s for (d,v) in zip(domain,exps)}
    else:
        return [v/s for v in exps]

def indicator(v, domain):
    return [1 if v==dv else 0 for dv in domain]
```

The following tests the learner on a datasets. Uncomment another dataset for different examples.

learnLinear.py — (continued)

```
from learnProblem import Data_set, Data_from_file, Evaluate
from learnProblem import Evaluate
import matplotlib.pyplot as plt

if __name__ == "__main__":
    data = Data_from_file('data/SPECT.csv', target_index=0)
    # data = Data_from_file('data/mail_reading.csv', target_index=-1)
    # data = Data_from_file('data/carbool.csv', one_hot=True,
        target_index=-1)
    Linear_learner(data).evaluate()
```

The following plots the errors on the training and validation sets as a function of the number of steps of gradient descent.

learnLinear.py — (continued)

```
def plot_steps(data,
               learner=None,
               criterion=Evaluate.squared_loss,
               step=1,
               num_steps=1000,
               log_scale=True,
               legend_label=""):
    """
    plots the training and validation error for a learner.
    data is the dataset
    learner is the learning algorithm (default is linear learner on the
        data)
    criterion gives the evaluation criterion plotted on the y-axis
    step specifies how many steps are run for each point on the plot
    num_steps is the number of points to plot
```

```
121
122     """
123     if legend_label != "": legend_label+=" "
124     plt.ion()
125     fig, ax = plt.subplots()
126     ax.set_xlabel("step")
127     ax.set_ylabel("Average "+criterion.__doc__)
128     if log_scale:
129         ax.set_xscale('log') #plt.semilogx() #Makes a log scale
130     else:
131         ax.set_xscale('linear')
132     if learner is None:
133         learner = Linear_learner(data)
134     train_errors = []
135     valid_errors = []
136     for i in range(1,num_steps+1,step):
137         valid_errors.append(data.evaluate_dataset(data.valid,
                learner.predictor, criterion))
138         train_errors.append(data.evaluate_dataset(data.train,
                learner.predictor, criterion))
139         learner.display(2, "Train error:",train_errors[-1],
140                         "Valid error:",valid_errors[-1])
141         learner.learn(num_iter=step)
142     ax.plot(range(1,num_steps+1,step),train_errors,ls='-',label=legend_label+"training")
143     ax.plot(range(1,num_steps+1,step),valid_errors,ls='--',label=legend_label+"validation")
144     ax.legend()
145     #plt.draw()
146     learner.display(1, "Train error:",train_errors[-1],
147                         "Validation error:",valid_errors[-1])
148
149 # This generates the figure
150 # from learnProblem import Data_set_augmented, prod_feat
151 # data = Data_from_file('data/SPECT.csv', prob_valid=0.5, target_index=0,
        seed=123)
152 # dataplus = Data_set_augmented(data, [], [prod_feat])
153 # plot_steps(data, num_steps=1000)
154 # plot_steps(dataplus, num_steps=1000) # warning slow
```

Figure 7.3 shows the result of `plot_steps(data, num_steps=1000)` in the code above. What would you expect to happen with the augmented data (with extra features)? Hint: think about underfitting and overfitting.

Exercise 7.12 In Figure 7.3, the log loss is very unstable when there are over 20 steps. Hypothesize why this occurs. [Hint: when does gradient descent become unstable?] Test your hypothesis by running with different hyperparameters.

Exercise 7.13 The squashed learner only makes predictions in the range $(0,1)$. If the output values are $\{1,2,3,4\}$ there is no use predicting less than 1 or greater than 4. Change the squashed learner so that it can learn values in the range $(1,4)$. Test it on the file `'data/car.csv'`.

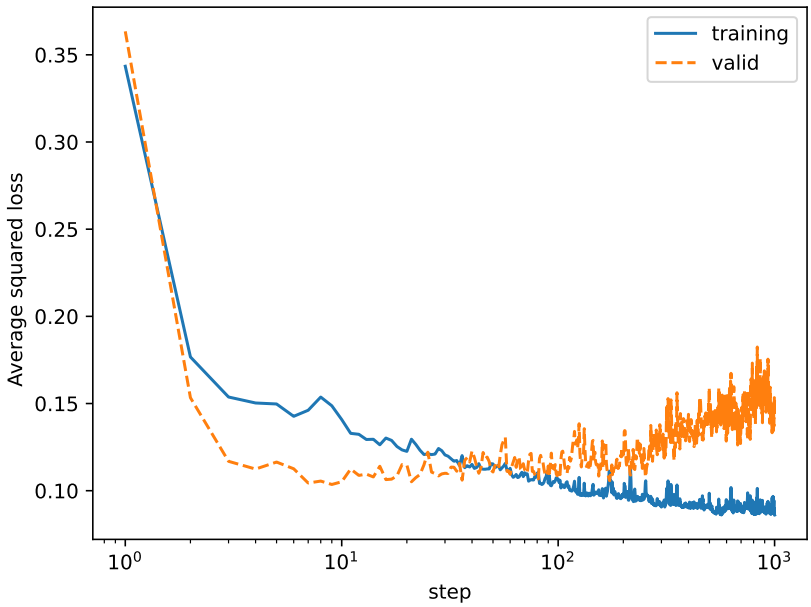


Figure 7.3: plot_steps for SPECT dataset

The following plots the prediction as a function of the number of steps of gradient descent. We first define a version of *range* that allows for real numbers (integers and floats). This is similar to numpy.arange.

learnLinear.py — (continued)

```
156 def arange(start,stop,step):
157     """enumerates values in the range [start,stop) separated by step.
158     like range(start,stop,step) but allows for integers and floats.
159     Rounding errors are expected with real numbers. (or use numpy.arange)
160     """
161     while start<stop:
162         yield start
163         start += step
164 
165 def plot_prediction(data,
166                 learner = None,
167                 minx = 0,
168                 maxx = 5,
169                 step_size = 0.01,  # for plotting
170                 label = "function"):
171     plt.ion()
172     fig,ax = plt.subplots()
173     ax.set_xlabel("x")
174     ax.set_ylabel("y")
```

```
175     if learner is None:
176         learner = Linear_learner(data, squashed=False)
177     learner.learning_rate=0.001
178     learner.learn(num_iter=100)
179     learner.learning_rate=0.0001
180     learner.learn(num_iter=1000)
181     learner.learning_rate=0.00001
182     learner.learn(num_iter=10000)
183     learner.display(1,f"function learned is {learner}. "
184              "error=",data.evaluate_dataset(data.train, learner.predictor,
                     Evaluate.squared_loss))
185     ax.plot([e[0] for e in data.train],[e[-1] for e in
            data.train],"bo",label="data")
186     ax.plot(list(arange(minx,maxx,step_size)),
187             [learner.predictor([x])
188                for x in arange(minx,maxx,step_size)],
189             label=label)
190     ax.legend(loc='upper left')
```

learnLinear.py — (continued)

```
192 from learnProblem import Data_set_augmented, power_feat
193 def plot_polynomials(data,
194                  learner_class = Linear_learner,
195                  max_degree = 5,
196                  minx = 0,
197                  maxx = 5,
198                  num_iter = 1000000,
199                  learning_rate = 0.00001,
200                  step_size = 0.01, # for plotting
201                  ):
202     plt.ion()
203     fig, ax = plt.subplots()
204     ax.set_xlabel("x")
205     ax.set_ylabel("y")
206     ax.plot([e[0] for e in data.train],[e[-1] for e in
            data.train],"ko",label="data")
207     x_values = list(arange(minx,maxx,step_size))
208     line_styles = ['-','--','-.',':']
209     colors = ['0.5','k','k','k','k']
210     for degree in range(max_degree):
211         data_aug = Data_set_augmented(data,[power_feat(n) for n in
                range(1,degree+1)],
212                                       include_orig=False)
213         learner = learner_class(data_aug,squashed=False)
214         learner.learning_rate = learning_rate
215         learner.learn(num_iter=num_iter)
216         learner.display(1,f"For degree {degree}, "
217                         f"function learned is {learner}. "
218                         "error=",data.evaluate_dataset(data.train,
                                learner.predictor, Evaluate.squared_loss))
```

```
        ls = line_styles[degree % len(line_styles)]
        col = colors[degree % len(colors)]
        ax.plot(x_values,[learner.predictor([x]) for x in x_values],
            linestyle=ls, color=col,
                        label="degree="+str(degree))
        ax.legend(loc='upper left')

# Try:
# data0 = Data_from_file('data/simp_regr.csv', prob_test=0, prob_valid=0,
    one_hot=False, target_index=-1)
# plot_prediction(data0)
# Alternatively:
# plot_polynomials(data0)
# What if the step size was bigger?
#datam = Data_from_file('data/mail_reading.csv', target_index=-1)
#plot_prediction(datam)
```

Exercise 7.14 For each of the polynomial functions learned: What is the prediction as x gets larger ($x \to \infty$). What is the prediction as x gets more negative ($x \to -\infty$).

7.7 Boosting

The following code implements functional gradient boosting for regression.

A Boosted dataset is created from a base dataset by subtracting the prediction of the offset function from each example. This does not save the new dataset, but generates it as needed. The extra space used is constant, independent on the size of the dataset.

learnBoosting.py — Functional Gradient Boosting

```
from learnProblem import Data_set, Learner, Evaluate
from learnNoInputs import Predict
from learnLinear import sigmoid
import statistics
import random

class Boosted_dataset(Data_set):
    def __init__(self, base_dataset, offset_fun, subsample=1.0):
        """new dataset which is like base_dataset,
           but offset_fun(e) is subtracted from the target of each example e
        """
        self.base_dataset = base_dataset
        self.offset_fun = offset_fun
        self.train =
            random.sample(base_dataset.train,int(subsample*len(base_dataset.train)))
        self.valid = base_dataset.valid
        #Data_set.__init__(self, base_dataset.train, base_dataset.valid,
        #                  base_dataset.prob_valid, base_dataset.target_index)

```

```
29          #def create_features(self):
30          """creates new features - called at end of Data_set.init()
31          defines a new target
32          """
33          self.input_features = self.base_dataset.input_features
34          def newout(e):
35              return self.base_dataset.target(e) - self.offset_fun(e)
36          newout.frange = self.base_dataset.target.frange
37          newout.ftype = self.infer_type(newout.frange)
38          self.target = newout
39
40      def conditions(self, *args, colsample_bytree=0.5, **nargs):
41          conds = self.base_dataset.conditions(*args, **nargs)
42          return random.sample(conds, int(colsample_bytree*len(conds)))
```

A boosting learner takes in a dataset and a base learner, and returns a new predictor. The base learner, takes a dataset, and returns a Learner object.

learnBoosting.py — (continued)

```
44  class Boosting_learner(Learner):
45      def __init__(self, dataset, base_learner_class, subsample=0.8):
46          self.dataset = dataset
47          self.base_learner_class = base_learner_class
48          self.subsample = subsample
49          mean = sum(self.dataset.target(e)
50                     for e in self.dataset.train)/len(self.dataset.train)
51          self.predictor = lambda e:mean  # function that returns mean for
                each example
52          self.predictor.__doc__ = "lambda e:"+str(mean)
53          self.offsets = [self.predictor] # list of base learners
54          self.predictors = [self.predictor] # list of predictors
55          self.errors = [data.evaluate_dataset(data.valid, self.predictor,
                Evaluate.squared_loss)]
56          self.display(1,"Mean validation set squared loss=", self.errors[0] )
57
58
59      def learn(self, num_ensembles=10):
60          """adds num_ensemble learners to the ensemble.
61          returns a new predictor.
62          """
63          for i in range(num_ensembles):
64              train_subset = Boosted_dataset(self.dataset, self.predictor,
                    subsample=self.subsample)
65              learner = self.base_learner_class(train_subset)
66              new_offset = learner.learn()
67              self.offsets.append(new_offset)
68              def new_pred(e, old_pred=self.predictor, off=new_offset):
69                  return old_pred(e)+off(e)
70              self.predictor = new_pred
71              self.predictors.append(new_pred)
```

```
72            self.errors.append(data.evaluate_dataset(data.valid,
                  self.predictor, Evaluate.squared_loss))
73            self.display(1,f"Iteration {len(self.offsets)-1},treesize =
                  {new_offset.num_leaves}. mean squared
                  loss={self.errors[-1]}")
74        return self.predictor
```

For testing, *sp_DT_learner* returns a learner that predicts the mean at the leaves and is evaluated using squared loss. It can also take arguments to change the default arguments for the trees.

learnBoosting.py — (continued)

```
76  # Testing
77
78  from learnDT import DT_learner
79  from learnProblem import Data_set, Data_from_file
80
81  def sp_DT_learner(split_to_optimize=Evaluate.squared_loss,
82                            leaf_prediction=Predict.mean,**nargs):
83      """Creates a learner with different default arguments replaced by
            **nargs
84      """
85      def new_learner(dataset):
86          return DT_learner(dataset,split_to_optimize=split_to_optimize,
87                                leaf_prediction=leaf_prediction, **nargs)
88      return new_learner
89
90  #data = Data_from_file('data/car.csv', target_index=-1) regression
91  #data = Data_from_file('data/SPECT.csv', target_index=0, seed=62) #123)
92  #data = Data_from_file('data/mail_reading.csv', target_index=-1)
93  #data = Data_from_file('data/holiday.csv', has_header=True, num_train=19,
        target_index=-1)
94  #learner10 = Boosting_learner(data,
        sp_DT_learner(split_to_optimize=Evaluate.squared_loss,
        leaf_prediction=Predict.mean, min_child_weight=10))
95  #learner7 = Boosting_learner(data, sp_DT_learner(0.7))
96  #learner5 = Boosting_learner(data, sp_DT_learner(0.5))
97  #predictor9 =learner9.learn(10)
98  #for i in learner9.offsets: print(i.__doc__)
99  import matplotlib.pyplot as plt
100
101 def plot_boosting_trees(data, steps=10, mcws=[30,20,20,10], gammas=
        [100,200,300,500]):
102     # to reduce clutter uncomment one of following two lines
103     #mcws=[10]
104     #gammas=[200]
105     learners = [(mcw, gamma, Boosting_learner(data,
            sp_DT_learner(min_child_weight=mcw, gamma=gamma)))
106                 for gamma in gammas for mcw in mcws
107                 ]
108     plt.ion()
```

```
    fig, ax = plt.subplots()
    ax.set_xscale('linear') # change between log and linear scale
    ax.set_xlabel("number of trees")
    ax.set_ylabel("mean squared loss")
    markers = (m+c for c in ['k','g','r','b','m','c','y'] for m in
        ['-','--','-.',':'])
    for (mcw,gamma,learner) in learners:
        data.display(1,f"min_child_weight={mcw}, gamma={gamma}")
        learner.learn(steps)
        ax.plot(range(steps+1), learner.errors, next(markers),
                    label=f"min_child_weight={mcw}, gamma={gamma}")
    ax.legend()

# plot_boosting_trees(data,mcws=[20], gammas= [100,200,300,500])
# plot_boosting_trees(data,mcws=[30,20,20,10], gammas= [100])
```

Exercise 7.15 For a particular dataset, suggest good values for `min_child_weight` and `gamma`. How stable are these to different random choices that are made (e.g., in the training-validation split)? Try to explain why these are good settings.

7.7.1 Gradient Tree Boosting

The following implements gradient Boosted trees for classification. If you want to use this gradient tree boosting for a real problem, we recommend using **XGBoost** [Chen and Guestrin, 2016] or **LightGBM** [Ke, Meng, Finley, Wang, Chen, Ma, Ye, and Liu, 2017].

`GTB_learner` subclasses `DT_learner`. The method `learn_tree` is used unchanged. `DT_learner` assumes that the value at the leaf is the prediction of the leaf, thus `leaf_value` needs to be overridden. It also assumes that all nodes at a leaf have the same prediction, but in GBT the elements of a leaf can have different values, depending on the previous trees. Thus `sum_losses` also needs to be overridden.

learnBoosting.py — (continued)

```
class GTB_learner(DT_learner):
    def __init__(self, dataset, number_trees, lambda_reg=1, gamma=0,
        **dtargs):
        DT_learner.__init__(self, dataset,
            split_to_optimize=Evaluate.log_loss, **dtargs)
        self.number_trees = number_trees
        self.lambda_reg = lambda_reg
        self.gamma = gamma
        self.trees = []

    def learn(self):
        for i in range(self.number_trees):
            tree =
                self.learn_tree(self.dataset.conditions(self.max_num_cuts),
                self.train)
```

```
135             self.trees.append(tree)
136             self.display(1,f"""Iteration {i} treesize = {tree.num_leaves}
                    train logloss={
137                 self.dataset.evaluate_dataset(self.dataset.train,
                        self.gtb_predictor, Evaluate.log_loss)
138                     } validation logloss={
139                 self.dataset.evaluate_dataset(self.dataset.valid,
                        self.gtb_predictor, Evaluate.log_loss)}""")
140         return self.gtb_predictor
141
142     def gtb_predictor(self, example, extra=0):
143         """prediction for example,
144         extras is an extra contribution for this example being considered
145         """
146         return sigmoid(sum(t(example) for t in self.trees)+extra)
147
148     def leaf_value(self, egs, domain=[0,1]):
149         """value at the leaves for examples egs
150         domain argument is ignored"""
151         pred_acts = [(self.gtb_predictor(e),self.target(e)) for e in egs]
152         return sum(a-p for (p,a) in pred_acts) /(sum(p*(1-p) for (p,a) in
                pred_acts)+self.lambda_reg)
153
154
155     def sum_losses(self, data_subset):
156         """returns sum of losses for dataset (assuming a leaf is formed
                with no more splits)
157         """
158         leaf_val = self.leaf_value(data_subset)
159         error = sum(Evaluate.log_loss(self.gtb_predictor(e,leaf_val),
                self.target(e))
160                 for e in data_subset) + self.gamma
161         return error
```

Testing

learnBoosting.py — (continued)

```
163 # data = Data_from_file('data/carbool.csv', one_hot=True, target_index=-1,
        seed=123)
164 # gtb_learner = GTB_learner(data, 10)
165 # gtb_learner.learn()
```

Exercise 7.16 Find better hyperparameter settings than the default ones. Compare prediction error with other methods for Boolean datasets.

Chapter 8

Neural Networks and Deep Learning

Warning: this is not meant to be an efficient implementation of deep learning. If you want to do serious machine learning on medium-sized or large data, we recommend Keras (`https://keras.io`) [Chollet, 2021] or PyTorch (`https://pytorch.org`), which are very efficient, particularly on GPUs. They are, however, black boxes. The AIPython neural network code should be seen like a car engine made of glass; you can see exactly how it works, even if it is not fast.

We have followed the naming conventions of Keras for the parameters: any parameters that are the same as in Keras have the same names.

8.1 Layers

A neural network is built from layers. In AIPython (unlike Keras and PyTorch), activation functions are treated as separate layers, which makes them more modular and the code more readable.

This provides a modular implementation of layers. Layers can easily be stacked in many configurations. A layer needs to implement a method to compute the output values from the inputs, a method to back-propagate the error, and a method update its parameters (if it has any) for a batch.

learnNN.py — Neural Network Learning

```
11 from display import Displayable
12 from learnProblem import Learner, Data_set, Data_from_file,
       Data_from_files, Evaluate
13 from learnLinear import sigmoid, one, softmax, indicator
14 import random, math, time
15
```

```
class Layer(Displayable):
    def __init__(self, nn, num_outputs=None):
        """Abstract layer class, must be overridden.
        nn is the neural network this layer is part of
        num outputs is the number of outputs for this layer.
        """
        self.nn = nn
        self.num_inputs = nn.num_outputs # nn output is layer's input
        if num_outputs:
            self.num_outputs = num_outputs
        else:
            self.num_outputs = self.num_inputs # same as the inputs
        self.outputs= [0]*self.num_outputs
        self.input_errors = [0]*self.num_inputs
        self.weights = []

    def output_values(self, input_values, training=False):
        """Return the outputs for this layer for the given input values.
        input_values is a list (of length self.num_inputs) of the inputs
        returns a list of length self.num_outputs.
        It can act differently when training and when predicting.
        """
        raise NotImplementedError("output_values") # abstract method

    def backprop(self, out_errors):
        """Backpropagate the errors on the outputs
        errors is a list of output errors (of length self.num_outputs).
        Returns list of input errors (of length self.num_inputs).

        This is only called after corresponding output_values(),
           which should remember relevant information
        """
        raise NotImplementedError("backprop") # abstract method

class Optimizer(Displayable):
    def update(self, layer):
        """updates parameters after a batch.
        """
        pass
```

8.1.1 Linear Layer

A linear layer maintains an array of weights. `self.weights[i][o]` is the weight between input `i` and output `o`. The bias is treated implicitly as the last input, so the weight of the bias for output `o` is `self.weights[self.num_inputs][o]`.

The default initialization is the Glorot uniform initializer [Glorot and Bengio, 2010], which is the default in Keras. An alternative is to provide a limit, in which case the values are selected uniformly in the range $[-limit, limit]$. As in Keras, AIpython treats initializes the bias of hidden layers to zero. The out-

put layer is treated separately, with the weights all zero except for the bias for categorical outputs (see following exercise).

learnNN.py — (continued)

```
class Linear_complete_layer(Layer):
    """a completely connected layer"""
    def __init__(self, nn, num_outputs, limit=None, final_layer=False):
        """A completely connected linear layer.
        nn is a neural network that the inputs come from
        num_outputs is the number of outputs
        the random initialization of parameters is in range [-limit,limit]
        """
        Layer.__init__(self, nn, num_outputs)
        if limit is None:
            limit =math.sqrt(6/(self.num_inputs+self.num_outputs))
        # self.weights[i][o] is the weight between input i and output o
        if final_layer:
             self.weights = [[0 if i < self.num_inputs
                                 or (nn.output_type != "categorical")
                               else 1
                            for o in range(self.num_outputs)]
                        for i in range(self.num_inputs+1)]
        else:
            self.weights = [[random.uniform(-limit, limit)
                            if i < self.num_inputs else 0
                            for o in range(self.num_outputs)]
                        for i in range(self.num_inputs+1)]
        # self.weights[i][o] is the accumulated change for a batch.
        self.delta = [[0 for o in range(self.num_outputs)]
                        for i in range(self.num_inputs+1)]

    def output_values(self, inputs, training=False):
        """Returns the outputs for the input values.
        It remembers the values for the backprop.
        """
        self.display(3,f"Linear layer inputs: {inputs}")
        self.inputs = inputs
        for out in range(self.num_outputs):
            self.outputs[out] = (sum(self.weights[inp][out]*self.inputs[inp]
                                     for inp in range(self.num_inputs))
                                 + self.weights[self.num_inputs][out])
        self.display(3,f"Linear layer inputs: {inputs}")
        return self.outputs

    def backprop(self, errors):
        """Backpropagate errors, update weights, return input error.
        errors is a list of size self.num_outputs
        Returns errors for layer's inputs of size
        """
        self.display(3,f"Linear Backprop. input: {self.inputs} output
            errors: {errors}")
```

```
        for out in range(self.num_outputs):
            for inp in range(self.num_inputs):
                self.input_errors[inp] = self.weights[inp][out] * errors[out]
                self.delta[inp][out] += self.inputs[inp] * errors[out]
            self.delta[self.num_inputs][out] += errors[out]
        self.display(3,f"Linear layer backprop input errors:
            {self.input_errors}")
        return self.input_errors
```

Exercise 8.1 The initialization for the output layer is naive. Suggest an alternative (hopefully better) initialization. Test it.

Exercise 8.2 What happens if the initialization of the hidden layer weights is also zero? Try it. Explain why you get the behavior observed.

8.1.2 ReLU Layer

The standard activation function for hidden nodes is the **ReLU**.

learnNN.py — (continued)

```
class ReLU_layer(Layer):
    """Rectified linear unit (ReLU) f(z) = max(0, z).
    The number of outputs is equal to the number of inputs.
    """
    def __init__(self, nn):
        Layer.__init__(self, nn)


    def output_values(self, input_values, training=False):
        """Returns the outputs for the input values.
        It remembers the input values for the backprop.
        """
        self.input_values = input_values
        for i in range(self.num_inputs):
            self.outputs[i] = max(0,input_values[i])
        return self.outputs

    def backprop(self,out_errors):
        """Returns the derivative of the errors"""
        for i in range(self.num_inputs):
            self.input_errors[i] = out_errors[i] if self.input_values[i]>0
                else 0
        return self.input_errors
```

8.1.3 Sigmoid Layer

One of the old standards for the activation function for hidden layers is the sigmoid. It is also used in LSTMs. It is included here to experiment with.

learnNN.py — (continued)

```
class Sigmoid_layer(Layer):
    """sigmoids of the inputs.
    The number of outputs is equal to the number of inputs.
    Each output is the sigmoid of its corresponding input.
    """
    def __init__(self, nn):
        Layer.__init__(self, nn)

    def output_values(self, input_values, training=False):
        """Returns the outputs for the input values.
        It remembers the output values for the backprop.
        """
        for i in range(self.num_inputs):
            self.outputs[i] = sigmoid(out_errors[i])
        return self.outputs

    def backprop(self,errors):
        """Returns the derivative of the errors"""
        for i in range(self.num_inputs):
            self.input_errors[i] =
                input_values[i]*out_errors[i]*(1-out_errors[i])
        return self.input_errors
```

8.2 Feedforward Networks

learnNN.py — (continued)

```
class NN(Learner):
    def __init__(self, dataset, optimizer=None, **hyperparms):
        """Creates a neural network for a dataset
        optimizer is the optimizer: default is SGD
        hyperparms is the dictionary of hyperparameters for the optimizer
        """
        self.dataset = dataset
        self.optimizer = optimizer if optimizer else SGD
        self.hyperparms = hyperparms
        self.output_type = dataset.target.ftype
        self.input_features = dataset.input_features
        self.num_outputs = len(self.input_features) # empty NN
        self.layers = []
        self.bn = 0 # number of batches run
        self.printed_heading = False

    def add_layer(self,layer):
        """add a layer to the network.
        Each layer gets number of inputs from the previous layers outputs.
        """
        self.layers.append(layer)
```

```
176         #if hasattr(layer, 'weights'):
177         layer.optimizer = self.optimizer(layer, **self.hyperparms)
178         self.num_outputs = layer.num_outputs
179 
180     def predictor(self,ex):
181         """Predicts the value of the first output for example ex.
182         """
183         values = [f(ex) for f in self.input_features]
184         for layer in self.layers:
185             values = layer.output_values(values)
186         return sigmoid(values[0]) if self.output_type =="boolean" \
187                else softmax(values, self.dataset.target.frange) if
                        self.output_type == "categorical" \
188                else values[0]
```

The *learn* method learns the parameters of a network. This is like the `learn()` method of linear regression (Section 7.6) except that there can be multiple outputs and there can be multiple optimizers.

learnNN.py — (continued)

```
190     def learn(self, batch_size=32, num_iter = 100, report_each=10):
191         """Learns parameters for a neural network using the chosen
                optimizer.
192         batch_size is the maximum size of each batch
193         num_iter is the number of iterations over the batches
194         report_each means print errors after each multiple of that number
                of batches
195         """
196         batch_size = min(batch_size, len(self.dataset.train)) # don't have
                batches bigger than training size
197         self.report_each = report_each
198         if not self.printed_heading and num_iter >= report_each:
199             self.display(1,"batch\tTraining\tTraining\tValidation\tValidation")
200             self.display(1,"\tAcccuracy\tLog loss\tAcccuracy\tLog loss")
201             self.printed_heading = True
202             self.trace()
203         for i in range(num_iter):
204             batch = random.sample(self.dataset.train, batch_size)
205             for e in batch:
206                 # compute all outputs
207                 values = [f(e) for f in self.input_features]
208                 for layer in self.layers:
209                     values = layer.output_values(values, training=True)
210                 # backpropagate
211                 predicted = [sigmoid(v) for v in values] \
212                             if self.output_type == "boolean" \
213                            else softmax(values) \
214                             if self.output_type == "categorical" \
215                            else values
216                 actuals = indicator(self.dataset.target(e),
                        self.dataset.target.frange) \
```

```
                        if self.output_type == "categorical"\
                        else [self.dataset.target(e)]
                errors = [pred-obsd for (obsd,pred) in
                    zip(actuals,predicted)]
                for layer in reversed(self.layers):
                    errors = layer.backprop(errors)
            # Update all parameters in batch
            for layer in self.layers:
                layer.optimizer.update(layer)
            self.bn+=1
            if (i+1)%report_each==0:
                self.trace()

    def trace(self):
        """print tracing of the batch updates"""
        self.display(1,self.bn,"\t",
            "\t\t".join("{:.4f}".format(
                    self.dataset.evaluate_dataset(data, self.predictor,
                        criterion))
                        for data in [self.dataset.train,
                            self.dataset.valid]
                        for criterion in [Evaluate.accuracy,
                            Evaluate.log_loss]), sep="")
```

8.3 Optimizers

The optimizers update the weights of a layer after a batch; they implement `update`. The layer must have saved the weights. In layers without weights, the weights list is empty, and update does nothing. The backprop method stores in `layer.delta` the gradient for the most recent batch. An optimizer must zero `layer.delta` so the new batch can start anew.

8.3.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is the most basic. It has one hyperparameter, the learning rate `lr`.

learnNN.py — (continued)

```
class SGD(Optimizer):
    """Vanilla SGD"""
    def __init__(self, layer, lr=0.01):
        """layer is a layer, which contains weight and gradient matrices
        Layers without weights have weights=[]
        """
        self.lr = lr

    def update(self, layer):
        """update weights of layer after a batch.
        """
```

```
248         for inp in range(len(layer.weights)):
249             for out in range(len(layer.weights[0])):
250                 layer.weights[inp][out] -= self.lr*layer.delta[inp][out]
251                 layer.delta[inp][out] = 0
```

8.3.2 Momentum

learnNN.py — (continued)

```
253 class Momentum(Optimizer):
254     """SGD with momentum"""
255 
256     """a completely connected layer"""
257     def __init__(self, layer, lr=0.01, momentum=0.9):
258         """
259         lr is the learning rate
260         momentum is the momentum parameter
261 
262         """
263         self.lr = lr
264         self.momentum = momentum
265         layer.velocity = [[0 for _ in range((len(layer.weights[0])))]
266                           for _ in range(len(layer.weights))]
267 
268 
269     def update(self, layer):
270         """updates parameters after a batch with momentum"""
271         for inp in range(len(layer.weights)):
272             for out in range(len(layer.weights[0])):
273                 layer.velocity[inp][out] =
                        self.momentum*layer.velocity[inp][out] -
                        self.lr*layer.delta[inp][out]
274                 layer.weights[inp][out] += layer.velocity[inp][out]
275                 layer.delta[inp][out] = 0
```

8.3.3 RMS-Prop

learnNN.py — (continued)

```
277 class RMS_Prop(Optimizer):
278     """a completely connected layer"""
279     def __init__(self, layer, rho=0.9, epsilon=1e-07, lr=0.01):
280         """A completely connected linear layer.
281         nn is a neural network that the inputs come from
282         num_outputs is the number of outputs
283         max_init is the maximum value for random initialization of
                parameters
284         """
285         # layer.ms[i][o] is running average of squared gradient input i and
                output o
```

```
286         layer.ms = [[0 for _ in range(len(layer.weights[0]))]
287                        for _ in range(len(layer.weights))]
288         self.rho = rho
289         self.epsilon = epsilon
290         self.lr = lr
291 
292     def update(self, layer):
293         """updates parameters after a batch"""
294         for inp in range(len(layer.weights)):
295             for out in range(len(layer.weights[0])):
296                 layer.ms[inp][out] = self.rho*layer.ms[inp][out]+
                        (1-self.rho) * layer.delta[inp][out]**2
297                 layer.weights[inp][out] -= self.lr * layer.delta[inp][out] /
                        (layer.ms[inp][out]+self.epsilon)**0.5
298                 layer.delta[inp][out] = 0
```

Exercise 8.3 Implement Adam [see Section 8.2.3 of Poole and Mackworth, 2023]. The implementation is slightly more complex than RMS-Prop. Try it first with the parameter settings of Keras, as reported by Poole and Mackworth [2023]. Does it matter if epsilon is inside or outside the square root? How sensitive is the performance to the parameter settings?

Exercise 8.4 Both Goodfellow, Bengio, and Courville [2016] and Poole and Mackworth [2023] find the gradient by dividing `self.delta[inp][out]` by the batch size, but some of the above code doesn't. To make code with dividing and without dividing the same, the step sizes need to be different by a factor of the batch size. Find a reasonable step size using an informal hyperparameter tuning; try some orders of magnitude of the step size to see what works best. What happens if the batch size is changed, but the step size is unchanged? (Try orders of magnitude difference is step sizes.) For each of the `update` method, which works better: dividing by the step size or not?

8.4 Dropout

Dropout is implemented as a layer.

learnNN.py — (continued)

```
300 from utilities import flip
301 class Dropout_layer(Layer):
302     """Dropout layer
303     """
304 
305     def __init__(self, nn, rate=0):
306         """
307         rate is fraction of the input units to drop. 0 =< rate < 1
308         """
309         self.rate = rate
310         Layer.__init__(self, nn)
311         self.mask = [0]*self.num_inputs
312 
```

```
313     def output_values(self, input_values, training=False):
314         """Returns the outputs for the input values.
315         It remembers the input values and mask for the backprop.
316         """
317         if training:
318             scaling = 1/(1-self.rate)
319             for i in range(self.num_inputs):
320                 self.mask[i] = 0 if flip(self.rate) else 1
321                 input_values[i] = self.mask[i]*input_values[i]*scaling
322         return input_values
323 
324     def backprop(self, output_errors):
325         """Returns the derivative of the errors"""
326         for i in range(self.num_inputs):
327             self.input_errors[i] = output_errors[i]*self.mask[i]
328         return self.input_errors
```

8.5 Examples

The following constructs some neural networks.

learnNN.py — (continued)

```
330 
331 def main():
332     """Sets up some global variables to allow for interaction
333     """
334     global data, nn3, nn3do
335     #data = Data_from_file('data/mail_reading.csv', target_index=-1)
336     #data = Data_from_file('data/mail_reading_consis.csv', target_index=-1)
337     data = Data_from_file('data/SPECT.csv', target_index=0) #, seed=12345)
338     #data = Data_from_file('data/carbool.csv', one_hot=True,
            target_index=-1, seed=123)
339     #data = Data_from_file('data/iris.data', target_index=-1)
340     #data = Data_from_file('data/if_x_then_y_else_z.csv', num_train=8,
            target_index=-1) # not linearly sep
341     #data = Data_from_file('data/holiday.csv', target_index=-1) #,
            num_train=19)
342     #data = Data_from_file('data/processed.cleveland.data', target_index=-1)
343     #random.seed(None)
344 
345     # nn3 is has a single hidden layer of width 3
346     nn3 = NN(data, optimizer=SGD)
347     nn3.add_layer(Linear_complete_layer(nn3,3))
348     #nn3.add_layer(Sigmoid_layer(nn3))
349     nn3.add_layer(ReLU_layer(nn3))
350     nn3.add_layer(Linear_complete_layer(nn3, 1, final_layer=True)) # when
            output_type="boolean"
351     print("nn3")
352     nn3.learn(batch_size=100, num_iter = 1000, report_each=100)
```

```
353
354     # Print some training examples
355     #for eg in random.sample(data.train,10): print(eg,nn3.predictor(eg))
356
357     # Print some test examples
358     #for eg in random.sample(data.test,10): print(eg,nn3.predictor(eg))
359
360     # To see the weights learned in linear layers
361     # nn3.layers[0].weights
362     # nn3.layers[2].weights
363
364     # nn3do is like nn3 but with dropout on the hidden layer
365     nn3do = NN(data, optimizer=SGD)
366     nn3do.add_layer(Linear_complete_layer(nn3do,3))
367     #nn3.add_layer(Sigmoid_layer(nn3)) # comment this or the next
368     nn3do.add_layer(ReLU_layer(nn3do))
369     nn3do.add_layer(Dropout_layer(nn3do, rate=0.5))
370     nn3do.add_layer(Linear_complete_layer(nn3do, 1, final_layer=True))
371     #nn3do.learn(batch_size=100, num_iter = 1000, report_each=100)
372
373 if __name__ == "__main__":
374     main()
```

`NN_from_arch(dataset, architecture, optimizer, parameters)` creates a generic feedforward neural network with ReLU activation for the hidden layers. The dataset is needed as the input and output is determined by the data. The architecture is a list of the sizes of hidden layers. If the architecture is the empty list, this corresponds to linear or logistic regression. The optimizer is one of `SGD`, `Momentum`, `RMS_Prop`.

learnNN.py — (continued)

```
376 class NN_from_arch(NN):
377     def __init__(self, data, arch, optimizer=SGD, **hyperparms):
378         """arch is a list of widths of the hidden layers from bottom up.
379         opt is an optimizer (one of: SGD, Momentum, RMS_Prop)
380         hyperparms is the parameters of the optimizer
381         returns a neural network with ReLU activations on hidden layers
382         """
383         NN.__init__(self, data, optimizer=optimizer, **hyperparms)
384         for width in arch:
385             self.add_layer(Linear_complete_layer(self,width))
386             self.add_layer(ReLU_layer(self))
387         output_size = len(data.target.frange) if data.target.ftype ==
                "categorical" else 1
388         self.add_layer(Linear_complete_layer(self,output_size,
                final_layer=True))
389         hyperparms_string = ','.join(f"{p}={v}" for p,v in
                hyperparms.items())
390         self.name = f"NN({arch},{optimizer.__name__}({hyperparms_string}))"
391
392     def __str__(self):
```

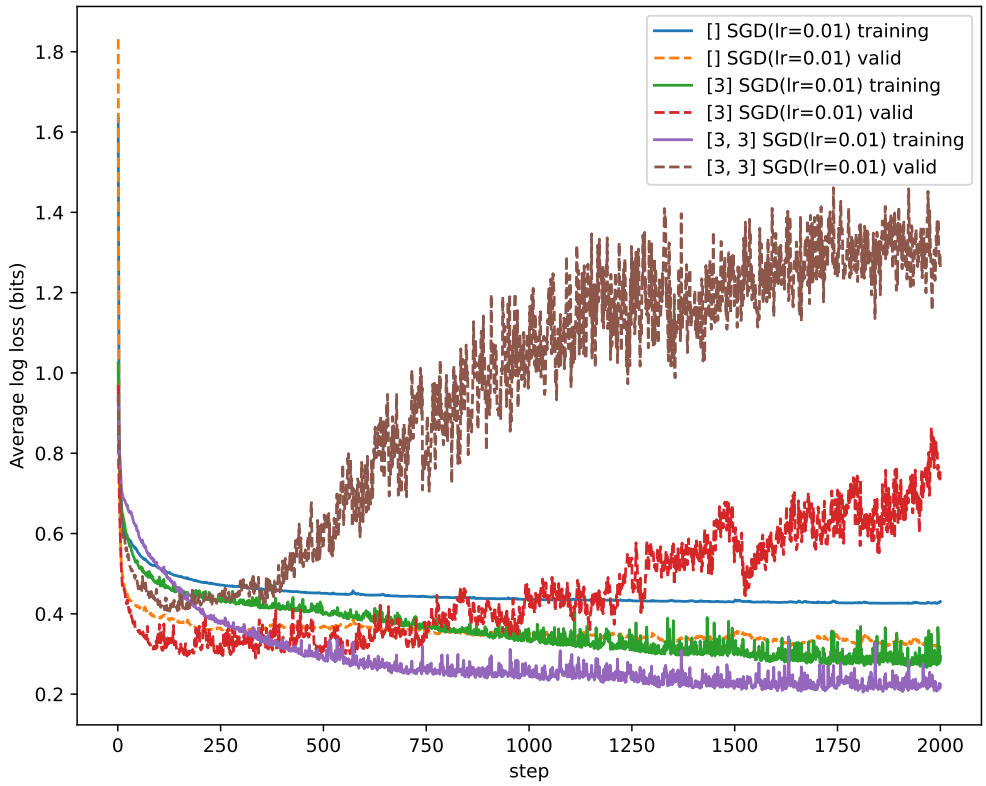


Figure 8.1: Plotting train and validation log loss for various architectures on SPECT dataset. Generated by `plot_algs(archs=[[],[3],[3,3]], opts=[SGD],lrs=[0.01],num_steps=2000)` Other runs might be different, as the validation set and the algorithm are stochastic.

```
393         return self.name
394
395 # nn3a = NN_from_arch(data, [3], SGD, lr=0.001)
```

8.6 Plotting Performance

You can plot the performance of various algorithms on the training and validation sets.

Figure 8.1 shows the training and validation performance on the SPECT dataset for the architectures given. The legend give the architecture, the optimizer, the options, and the evaluation dataset. The architecture [] is for logistic regression. Notice how, as the network gets larger the better they fit the training data, but can overfit more as the number of steps increases (probably because the probabilities get more extreme). These figures suggest that early stopping after 200-300 steps might provide best test performance.

The plot_algs method does all combinations of architectures, optimizers and learning rates. It plots both learning and validation errors. The output is only readable if two of these are singletons, and one varies (as in the examples).

The plot_algs_opts method is more general as it allows for different combinations of architectures, optimizers and learning rates, which makes more sense if, for example, the learning rate is set depending on the architecture and optimizer. It also allows other hyperparameters to be specified and varied.

learnNN.py — (continued)

```
from learnLinear import plot_steps
from learnProblem import Evaluate

# To show plots first choose a criterion to use
crit = Evaluate.log_loss # penalizes overconfident predictions (when wrong)
# crit = Evaluate.accuracy # only considers mode
# crit = Evaluate.squared_loss # penalizes overconfident predictions less

def plot_algs(data, archs=[[3]], opts=[SGD],lrs=[0.1, 0.01,0.001,0.0001],
                  criterion=crit, num_steps=1000):
    args = []
    for arch in archs:
        for opt in opts:
            for lr in lrs:
                args.append((arch,opt,{'lr':lr}))
    plot_algs_opts(data, args, criterion, num_steps)

def plot_algs_opts(data, args, criterion=crit, num_steps=1000):
    """args is a list of (architecture, optimizer, parameters)
       for each of the corresponding triples it plots the learning rate"""
    for (arch, opt, hyperparms) in args:
        nn = NN_from_arch(data, arch, opt, **hyperparms)
        plot_steps(data, learner = nn, criterion=crit, num_steps=num_steps,
                      log_scale=False, legend_label=str(nn))
```

The following are examples of how to do hyperparameter optimization manually.

learnNN.py — (continued)

```
## first select good learning rates for each optimizer.
# plot_algs(data, archs=[[3]], opts=[SGD],lrs=[0.1, 0.01,0.001,0.0001])
# plot_algs(data, archs=[[3]], opts=[Momentum],lrs=[0.1,
    0.01,0.001,0.0001])
# plot_algs(data, archs=[[3]], opts=[RMS_Prop],lrs=[0.1,
    0.01,0.001,0.0001])

## If they have the same best learning rate, compare the optimizers:
# plot_algs(data, archs=[[3]], opts=[SGD,Momentum,RMS_Prop],lrs=[0.01])

## With different learning rates, compare the optimizer using:
# plot_algs_opts(data, args=[([3],SGD,{'lr':0.01}),
    ([3],Momentum,{'lr':0.1}), ([3],RMS_Prop,{'lr':0.001})])
```

```
432
433 # similarly select the best architecture, but the best learning rate might
        depend also on the architecture
```

The following tests are on the MNIST digit dataset. The original files are from `http://yann.lecun.com/exdb/mnist/`. This code assumes you use the csv files from Joseph Redmon (`https://pjreddie.com/projects/mnist-in-csv/` or `https://github.com/pjreddie/mnist-csv-png` or `https://www.kaggle.com/datasets/oddrationale/mnist-in-csv`) and put them in the directory `../MNIST/`. Note that this is **very** inefficient; you would be better to use Keras or PyTorch. There are $28 * 28 = 784$ input units and 512 hidden units, which makes 401,408 parameters for the lowest linear layer. So don't be surprised if it takes many hours in AIPython (even if it only takes a few seconds in Keras).

Think about: with 10 classes what is the accuracy, absolute loss, squared loss, log loss (bits) for a naive guess (where the naive guess might depend on the criterion)?

learnNN.py — (continued)

```
435 # Simplified version: (approx 6000 training instances)
436 # data_mnist = Data_from_file('../MNIST/mnist_train.csv', prob_test=0.9,
        target_index=0, target_type="categorical")
437
438 # Full version:
439 # data_mnist = Data_from_files('../MNIST/mnist_train.csv',
        '../MNIST/mnist_test.csv', target_index=0, target_type="categorical")
440
441 #nn_mnist = NN_from_arch(data_mnist, [32,10], SGD, lr=0.01})
442 # one epoch:
443 # start_time = time.perf_counter();nn_mnist.learn(batch_size=128,
        num_iter=len(data_mnist)/128 );end_time =
        time.perf_counter();print("Time:", end_time - start_time,"seconds")
444 # determine train error:
445 # data_mnist.evaluate_dataset(data_mnist.train, nn_mnist.predictor,
        Evaluate.accuracy)
446 # determine test error:
447 # data_mnist.evaluate_dataset(data_mnist.test, nn_mnist.predictor,
        Evaluate.accuracy)
448 # Print some random predictions:
449 # for eg in random.sample(data_mnist.test,10):
        print(data_mnist.target(eg), nn_mnist.predictor(eg),
        nn_mnist.predictor(eg)[data_mnist.target(eg)])
450 # Plot learning:
451 # plot_algs(data_mnist,archs=[[32],[32,8]], opts=[RMS_Prop], lrs=[0.01],
        data=data_mnist, num_steps=100)
452 # plot_algs(data_mnist,archs=[[8],[8,8,8],[8,8,8,8,8,8,8]],
        opts=[RMS_Prop], lrs=[0.01], data=data_mnist, num_steps=100)
```

Exercise 8.5 In the definition of *nn*3 above, for each of the following, first hypothesize what will happen, then test your hypothesis, then explain whether you

testing confirms your hypothesis or not. Test it for more than one data set, and use more than one run for each data set.

(a) Which fits the data better, having a sigmoid layer or a ReLU layer after the first linear layer?

(b) Which is faster to learn, having a sigmoid layer or a ReLU layer after the first linear layer? (Hint: Plot error as a function of steps).

(c) What happens if you have both the sigmoid layer and then a ReLU layer after the first linear layer and before the second linear layer?

(d) What happens if you have a ReLU layer then a sigmoid layer after the first linear layer and before the second linear layer?

(e) What happens if you have neither the sigmoid layer nor a ReLU layer after the first linear layer?

Exercise 8.6 Select one dataset and architecture.

(a) For each optimizer, use the validation set to choose settings for the hyperparameters, including when to stop, and the parameters of the optimizer (including the learning rate). (There is no need to do an exhaustive search, and remember that the runs are stochastic.) For the dataset and architecture chosen, which optimizer works best?

(b) Suggest another architecture which you conjecture would be better than the one used in (a) on the test set (after hyperparameter optimization). Is it better?

Chapter 9

Reasoning with Uncertainty

9.1 Representing Probabilistic Models

A probabilistic model uses the same definition of a variable as a CSP (Section 4.1.1, page 69). A variable consists of a name, a domain and an optional (x,y) position (for displaying). The domain of a variable is a list or a tuple, as the ordering matters for some representation of factors.

9.2 Representing Factors

A **factor** is, mathematically, a function from variables into a number; that is, given a value for each of its variable, it gives a number. Factors are used for conditional probabilities, utilities in the next chapter, and are explicitly constructed by some algorithms (in particular, variable elimination).

A variable assignment, or just an **assignment**, is represented as a $\{variable : value\}$ dictionary. A factor can be evaluated when all of its variables are assigned. This is implemented in the `can_evaluate` method which can be overridden for representations that don't require all variable be assigned (such as decision trees). The method `get_value` evaluates the factor for an assignment. The assignment can include extra variables not in the factor. This method needs to be defined for every subclass.

probFactors.py — Factors for graphical models

```
11 from display import Displayable
12 import math
13
14 class Factor(Displayable):
15     nextid=0 # each factor has a unique identifier; for printing
16
```

```
17      def __init__(self, variables, name=None):
18          self.variables = variables # list of variables
19          if name:
20              self.name = name
21          else:
22              self.name = f"f{Factor.nextid}"
23              Factor.nextid += 1
24
25      def can_evaluate(self,assignment):
26          """True when the factor can be evaluated in the assignment
27          assignment is a {variable:value} dict
28          """
29          return all(v in assignment for v in self.variables)
30
31      def get_value(self,assignment):
32          """Returns the value of the factor given the assignment of values
                to variables.
33          Needs to be defined for each subclass.
34          """
35          assert self.can_evaluate(assignment)
36          raise NotImplementedError("get_value") # abstract method
```

The method `__str__` returns a brief definition (like "`f7(X,Y,Z)`").The method `to_table` returns string representations of a table showing all of the assignments of values to variables, and the corresponding value.

probFactors.py — (continued)

```
38      def __str__(self):
39          """returns a string representing a summary of the factor"""
40          return f"{self.name}({','.join(str(var) for var in
                self.variables)})"
41
42      def to_table(self, variables=None, given={}):
43          """returns a string representation of the factor.
44          Allows for an arbitrary variable ordering.
45          variables is a list of the variables in the factor
46          (can contain other variables)"""
47          if variables==None:
48              variables = [v for v in self.variables if v not in given]
49          else: #enforce ordering and allow for extra variables in ordering
50              variables = [v for v in variables if v in self.variables and v
                    not in given]
51          head = "\t".join(str(v) for v in variables)+"\t"+self.name
52          return head+"\n"+self.ass_to_str(variables, given, variables)
53
54      def ass_to_str(self, vars, asst, allvars):
55          #print(f"ass_to_str({vars}, {asst}, {allvars})")
56          if vars:
57              return "\n".join(self.ass_to_str(vars[1:], {**asst,
                        vars[0]:val}, allvars)
58                               for val in vars[0].domain)
```

```
59          else:
60              val = self.get_value(asst)
61              val_st = "{:.6f}".format(val) if isinstance(val,float) else
                    str(val)
62              return ("\t".join(str(asst[var]) for var in allvars)
63                        + "\t"+val_st)
64
65      __repr__ = __str__
```

9.3 Conditional Probability Distributions

A **conditional probability distribution (CPD)** is a factor that represents a conditional probability. A CPD representing $P(X \mid Y_1 \ldots Y_k)$ is a factor, which given values for X and each Y_i returns a number.

probFactors.py — (continued)

```
67  class CPD(Factor):
68      def __init__(self, child, parents):
69          """represents P(variable | parents)
70          """
71          self.parents = parents
72          self.child = child
73          Factor.__init__(self, parents+[child], name=f"Probability")
74
75      def __str__(self):
76          """A brief description of a factor using in tracing"""
77          if self.parents:
78              return f"P({self.child}|{','.join(str(p) for p in
                    self.parents)})"
79          else:
80              return f"P({self.child})"
81
82      __repr__ = __str__
```

A constant CPD has no parents, and has probability 1 when the variable has the value specified, and 0 when the variable has a different value.

probFactors.py — (continued)

```
84  class ConstantCPD(CPD):
85      def __init__(self, variable, value):
86          CPD.__init__(self, variable, [])
87          self.value = value
88      def get_value(self, assignment):
89          return 1 if self.value==assignment[self.child] else 0
```

9.3.1 Logistic Regression

A **logistic regression** CPD, for Boolean variable X represents $P(X{=}True \mid Y_1 \ldots Y_k)$, using $k+1$ real-valued weights so

$$P(X{=}True \mid Y_1 \ldots Y_k) = sigmoid(w_0 + \sum_i w_i Y_i)$$

where for Boolean Y_i, True is represented as 1 and False as 0.

probFactors.py — (continued)

```
91  from learnLinear import sigmoid, logit
92
93  class LogisticRegression(CPD):
94      def __init__(self, child, parents, weights):
95          """A logistic regression representation of a conditional
                probability.
96          child is the Boolean (or 0/1) variable whose CPD is being defined
97          parents is the list of parents
98          weights is list of parameters, such that weights[i+1] is the weight
                for parents[i]
99            weights[0] is the bias.
100         """
101         assert len(weights) == 1+len(parents)
102         CPD.__init__(self, child, parents)
103         self.weights = weights
104
105     def get_value(self,assignment):
106         assert self.can_evaluate(assignment)
107         prob = sigmoid(self.weights[0]
108                        + sum(self.weights[i+1]*assignment[self.parents[i]]
109                              for i in range(len(self.parents))))
110         if assignment[self.child]: #child is true
111             return prob
112         else:
113             return (1-prob)
```

9.3.2 Noisy-or

A **noisy-or**, for Boolean variable X with Boolean parents $Y_1 \ldots Y_k$ is parametrized by $k+1$ parameters $p_0, p_1, \ldots, p_k$, where each $0 \leq p_i \leq 1$. The semantics is defined as though there are $k+1$ hidden variables $Z_0, Z_1 \ldots Z_k$, where $P(Z_0) = p_0$ and $P(Z_i \mid Y_i) = p_i$ for $i \geq 1$, and where X is true if and only if $Z_0 \vee Z_1 \vee \cdots \vee Z_k$ (where $\vee$ is "or"). Thus X is false if all of the Z_i are false. Intuitively, Z_0 is the probability of X when all Y_i are false and each Z_i is a noisy (probabilistic) measure that Y_i makes X true, and X only needs one to make it true.

probFactors.py — (continued)

```
115 class NoisyOR(CPD):
116     def __init__(self, child, parents, weights):
```

```
117         """A noisy representation of a conditional probability.
118         variable is the Boolean (or 0/1) child variable whose CPD is being
                defined
119         parents is the list of Boolean (or 0/1) parents
120         weights is list of parameters, such that weights[i+1] is the weight
                for parents[i]
121         """
122         assert len(weights) == 1+len(parents)
123         CPD.__init__(self, child, parents)
124         self.weights = weights
125 
126     def get_value(self,assignment):
127         assert self.can_evaluate(assignment)
128         probfalse = (1-self.weights[0])*math.prod(1-self.weights[i+1]
129                                         for i in range(len(self.parents))
130                                           if assignment[self.parents[i]])
131         if assignment[self.child]: # child is assigned True in assignment
132             return 1-probfalse
133         else:
134             return probfalse
```

9.3.3 Tabular Factors and Prob

A **tabular factor** is a factor that represents each assignment of values to variables separately. It is represented by a Python array (or Python dict). If the variables are $V_1, V_2, \ldots, V_k$, the value of $f(V_1 = v_1, V_2 = v_1, \ldots, V_k = v_k)$ is stored in $f[v_1][v_2]\ldots[v_k]$.

If the domain of V_i is $[0, \ldots, n_i - 1]$ it can be represented as an array. Otherwise it can use a dictionary. Python is nice in that it doesn't care, whether an array or dict is used **except when enumerating the values**; enumerating a dict gives the keys (the variables) but enumerating an array gives the values. So we had to be careful not to enumerate the values.

probFactors.py — (continued)

```
136 class TabFactor(Factor):
137 
138     def __init__(self, variables, values, name=None):
139         Factor.__init__(self, variables, name=name)
140         self.values = values
141 
142     def get_value(self, assignment):
143         return self.get_val_rec(self.values, self.variables, assignment)
144 
145     def get_val_rec(self, value, variables, assignment):
146         if variables == []:
147            return value
148         else:
149             return self.get_val_rec(value[assignment[variables[0]]],
150                                     variables[1:],assignment)
```

Prob is a factor that represents a conditional probability by enumerating all of the values.

probFactors.py — (continued)

```
class Prob(CPD,TabFactor):
    """A factor defined by a conditional probability table"""
    def __init__(self, var, pars, cpt, name=None):
        """Creates a factor from a conditional probability table, cpt
        The cpt values are assumed to be for the ordering par+[var]
        """
        TabFactor.__init__(self, pars+[var], cpt, name)
        self.child = var
        self.parents = pars
```

9.3.4 Decision Tree Representations of Factors

A decision tree representation of a conditional probability of a child variable is either:

- `IFeq(var, val, true_cond, false_cond)` where `true_cond` and `false_cond` are decision trees. `true_cond` is used if variable `var` has value `val` in an assignment; `false_cond` is used if `var` has a different value
- a deterministic functions that has probability 1 if a parent has the same value as the child (using `SameAs(parent)`)
- a distribution over the child variable (using `Dist(dict)`).

Note that not all parents need to be assigned to evaluate the decision tree; it only needs a branch down the tree that gives the distribution.

probFactors.py — (continued)

```
class ProbDT(CPD):
    def __init__(self, child, parents, dt):
        CPD.__init__(self, child, parents)
        self.dt = dt

    def get_value(self, assignment):
        return self.dt.get_value(assignment, self.child)

    def can_evaluate(self, assignment):
        return self.child in assignment and self.dt.can_evaluate(assignment)
```

Decision trees are made up of conditions; here equality of a value and a variable:

probFactors.py — (continued)

```
class IFeq:
    def __init__(self, var, val, true_cond, false_cond):
        self.var = var
```

```
176         self.val = val
177         self.true_cond = true_cond
178         self.false_cond = false_cond
179 
180     def get_value(self, assignment, child):
181         """ IFeq(var, val, true_cond, false_cond)
182         value of true_cond is used if var has value val in assignment,
183         value of false_cond is used if var has a different value
184         """
185         if assignment[self.var] == self.val:
186             return self.true_cond.get_value(assignment, child)
187         else:
188             return self.false_cond.get_value(assignment,child)
189 
190     def can_evaluate(self, assignment):
191         if self.var not in assignment:
192             return False
193         elif assignment[self.var] == self.val:
194             return self.true_cond.can_evaluate(assignment)
195         else:
196             return self.false_cond.can_evaluate(assignment)
```

The following is a deterministic function that is true if the parent has the same value as the child. This is used for deterministic conditional probabilities (as is common for causal models, as described in Chapter 11).

probFactors.py — (continued)

```
198 class SameAs:
199     def __init__(self, parent):
200         """1 when child has same value as parent, otherwise 0"""
201         self.parent = parent
202 
203     def get_value(self, assignment, child):
204         return 1 if assignment[child]==assignment[self.parent] else 0
205 
206     def can_evaluate(self, assignment):
207         return self.parent in assignment
```

At the leaves are distributions over the child variable.

probFactors.py — (continued)

```
209 class Dist:
210     def __init__(self, dist):
211         """Dist is an array or dictionary indexed by value of current
                child"""
212         self.dist = dist
213 
214     def get_value(self, assignment, child):
215         return self.dist[assignment[child]]
216 
217     def can_evaluate(self, assignment):
218         return True
```

The following shows a decision representation of the Example 9.18 of Poole and Mackworth [2023]. When the `Action` is to go out, the probability is a function of `rain`; otherwise it is a function of `full`.

probFactors.py — (continued)

```
##### A decision tree representation Example 9.18 of AIFCA 3e
from variable import Variable

boolean = [False, True]

action = Variable('Action', ['go_out', 'get_coffee'], position=(0.5,0.8))
rain = Variable('Rain', boolean, position=(0.2,0.8))
full = Variable('Cup Full', boolean, position=(0.8,0.8))

wet = Variable('Wet', boolean, position=(0.5,0.2))
p_wet = ProbDT(wet,[action,rain,full],
                 IFeq(action, 'go_out',
                      IFeq(rain, True, Dist([0.2,0.8]), Dist([0.9,0.1])),
                      IFeq(full, True, Dist([0.4,0.6]), Dist([0.7,0.3]))))

# See probRC for wetBN which expands this example to a complete network
```

9.4 Graphical Models

A graphical model consists of a title, a set of variables, and a set of factors.

probGraphicalModels.py — Graphical Models and Belief Networks

```
from display import Displayable
from variable import Variable
from probFactors import CPD, Prob
import matplotlib.pyplot as plt

class GraphicalModel(Displayable):
    """The class of graphical models.
    A graphical model consists of a title, a set of variables and a set of
        factors.

    vars is a set of variables
    factors is a set of factors
    """
    def __init__(self, title, variables=None, factors=None):
        self.title = title
        self.variables = variables
        self.factors = factors
```

A **belief network** (also known as a **Bayesian network**) is a graphical model where all of the factors are conditional probabilities, and every variable has a conditional probability of it given its parents. This checks the first condi-

tion (that all factors are conditional probabilities), and builds some useful data structures.

probGraphicalModels.py — (continued)

```
class BeliefNetwork(GraphicalModel):
    """The class of belief networks."""

    def __init__(self, title, variables, factors):
        """vars is a set of variables
        factors is a set of factors. All of the factors are instances of
            CPD (e.g., Prob).
        """
        GraphicalModel.__init__(self, title, variables, factors)
        assert all(isinstance(f,CPD) for f in factors), factors
        self.var2cpt = {f.child:f for f in factors}
        self.var2parents = {f.child:f.parents for f in factors}
        self.children = {n:[] for n in self.variables}
        for v in self.var2parents:
            for par in self.var2parents[v]:
                self.children[par].append(v)
        self.topological_sort_saved = None
```

The following creates a topological sort of the nodes, where the parents of a node come before the node in the resulting order. This is based on Kahn's algorithm from 1962.

probGraphicalModels.py — (continued)

```
    def topological_sort(self):
        """creates a topological ordering of variables such that the
            parents of
        a node are before the node.
        """
        if self.topological_sort_saved:
            return self.topological_sort_saved
        next_vars = {n for n in self.var2parents if not self.var2parents[n]
            }
        self.display(3,'topological_sort: next_vars',next_vars)
        top_order=[]
        while next_vars:
            var = next_vars.pop()
            self.display(3,'select variable',var)
            top_order.append(var)
            next_vars |= {ch for ch in self.children[var]
                          if all(p in top_order for p in
                              self.var2parents[ch])}
            self.display(3,'var_with_no_parents_left',next_vars)
        self.display(3,"top_order",top_order)
        assert
            set(top_order)==set(self.var2parents),(top_order,self.var2parents)
        self.topologicalsort_saved=top_order
        return top_order
```

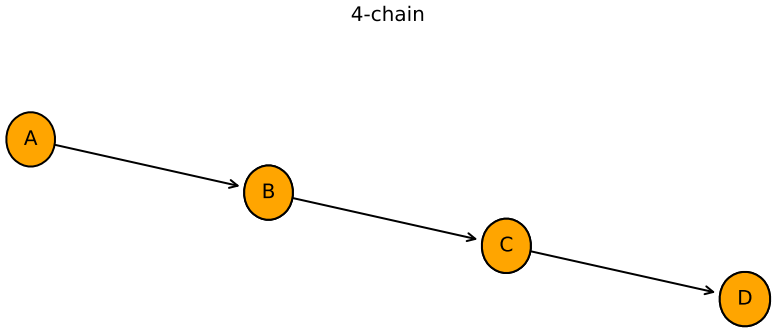


Figure 9.1: bn_4ch.show()

9.4.1 Showing Belief Networks

The **show** method uses matplotlib to show the graphical structure of a belief network.

probGraphicalModels.py — (continued)

```
66    def show(self, fontsize=10, facecolor='orange'):
67        plt.ion()  # interactive
68        fig, ax = plt.subplots()
69        ax.set_axis_off()
70        ax.set_title(self.title, fontsize=fontsize)
71        bbox =
              dict(boxstyle="round4,pad=1.0,rounding_size=0.5",facecolor=facecolor)
72        for var in self.variables: #reversed(self.topological_sort()):
73            for par in self.var2parents[var]:
74                   ax.annotate(var.name, par.position, xytext=var.position,
75                               arrowprops={'arrowstyle':'<-'},bbox=bbox,
76                               ha='center', va='center',
                                    fontsize=fontsize)
77        for var in self.variables:
78               x,y = var.position
79               ax.text(x,y,var.name,bbox=bbox,ha='center', va='center',
                     fontsize=fontsize)
```

9.4.2 Example Belief Networks

A Chain of 4 Variables

The first example belief network is a simple chain $A \longrightarrow B \longrightarrow C \longrightarrow D$, shown in Figure 9.1.

Please do not change this, as it is the example used for testing.

probGraphicalModels.py — (continued)

```
81 #### Simple Example Used for Unit Tests ####
```

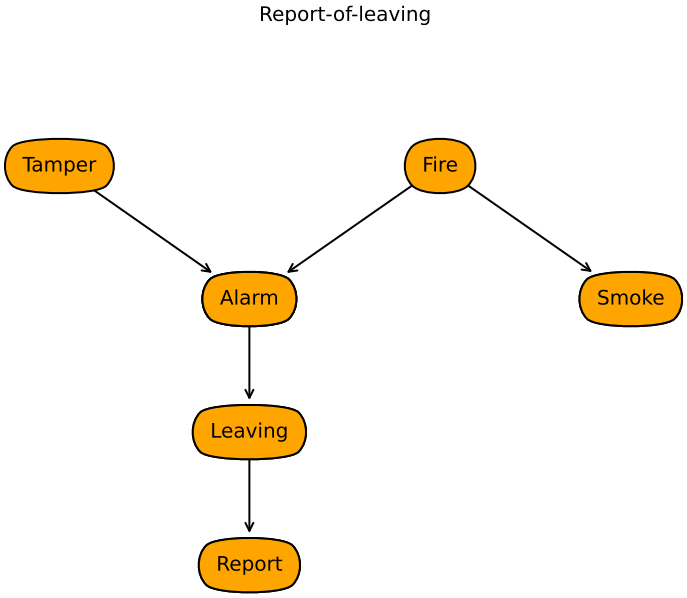


Figure 9.2: The report-of-leaving belief network

```
boolean = [False, True]
A = Variable("A", boolean, position=(0,0.8))
B = Variable("B", boolean, position=(0.333,0.7))
C = Variable("C", boolean, position=(0.666,0.6))
D = Variable("D", boolean, position=(1,0.5))

f_a = Prob(A,[],[0.4,0.6])
f_b = Prob(B,[A],[[0.9,0.1],[0.2,0.8]])
f_c = Prob(C,[B],[[0.6,0.4],[0.3,0.7]])
f_d = Prob(D,[C],[[0.1,0.9],[0.75,0.25]])

bn_4ch = BeliefNetwork("4-chain", {A,B,C,D}, {f_a,f_b,f_c,f_d})
```

Report-of-Leaving Example

The second belief network, `bn_report`, is Example 9.13 of Poole and Mackworth [2023] (`http://artint.info`). The output of `bn_report.show()` is shown in Figure 9.2 of this document.

probExamples.py — Example belief networks

```
from variable import Variable
from probFactors import CPD, Prob, LogisticRegression, NoisyOR, ConstantCPD
from probGraphicalModels import BeliefNetwork

```

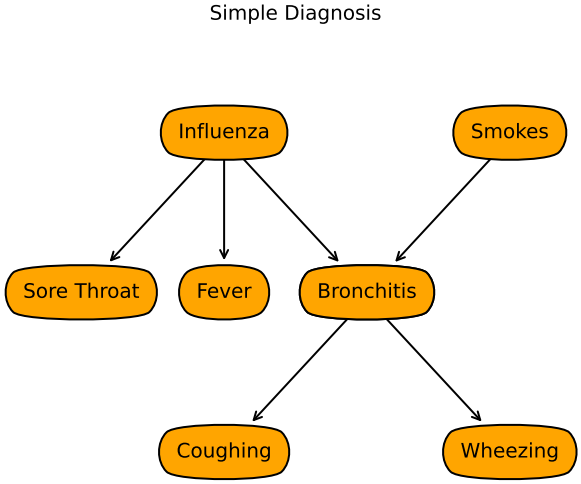


Figure 9.3: Simple diagnosis example; `simple_diagnosis.show()`

```
15 # Belief network report-of-leaving example (Example 9.13 shown in Figure
       9.3) of
16 # Poole and Mackworth, Artificial Intelligence, 2023 http://artint.info
17 boolean = [False, True]
18
19 Alarm =  Variable("Alarm", boolean, position=(0.366,0.5))
20 Fire =   Variable("Fire",  boolean, position=(0.633,0.75))
21 Leaving = Variable("Leaving", boolean, position=(0.366,0.25))
22 Report = Variable("Report", boolean, position=(0.366,0.0))
23 Smoke =  Variable("Smoke", boolean, position=(0.9,0.5))
24 Tamper = Variable("Tamper", boolean, position=(0.1,0.75))
25
26 f_ta = Prob(Tamper,[],[0.98,0.02])
27 f_fi = Prob(Fire,[],[0.99,0.01])
28 f_sm = Prob(Smoke,[Fire],[[0.99,0.01],[0.1,0.9]])
29 f_al = Prob(Alarm,[Fire,Tamper],[[[0.9999, 0.0001], [0.15, 0.85]], [[0.01,
       0.99], [0.5, 0.5]]])
30 f_lv = Prob(Leaving,[Alarm],[[0.999, 0.001], [0.12, 0.88]])
31 f_re = Prob(Report,[Leaving],[[0.99, 0.01], [0.25, 0.75]])
32
33 bn_report = BeliefNetwork("Report-of-leaving",
       {Tamper,Fire,Smoke,Alarm,Leaving,Report},
34                             {f_ta,f_fi,f_sm,f_al,f_lv,f_re})
```

Simple Diagnostic Example

This is the "simple diagnostic example" of Exercise 9.1 of Poole and Mackworth [2023], reproduced here as Figure 9.3

probExamples.py — (continued)

```
# Belief network simple-diagnostic example (Exercise 9.3 shown in Figure 9.39) of
# Poole and Mackworth, Artificial Intelligence, 2023 http://artint.info

Influenza =  Variable("Influenza", boolean, position=(0.4,0.8))
Smokes =     Variable("Smokes", boolean, position=(0.8,0.8))
SoreThroat = Variable("Sore Throat", boolean, position=(0.2,0.5))
HasFever =      Variable("Fever", boolean, position=(0.4,0.5))
Bronchitis = Variable("Bronchitis", boolean, position=(0.6,0.5))
Coughing =   Variable("Coughing", boolean, position=(0.4,0.2))
Wheezing =   Variable("Wheezing", boolean, position=(0.8,0.2))

p_infl =   Prob(Influenza,[],[0.95,0.05])
p_smokes = Prob(Smokes,[],[0.8,0.2])
p_sth =    Prob(SoreThroat,[Influenza],[[0.999,0.001],[0.7,0.3]])
p_fever =  Prob(HasFever,[Influenza],[[0.99,0.05],[0.9,0.1]])
p_bronc = Prob(Bronchitis,[Influenza,Smokes],[[[0.9999, 0.0001], [0.3, 0.7]], [[0.1, 0.9], [0.01, 0.99]]])
p_cough =  Prob(Coughing,[Bronchitis],[[0.93,0.07],[0.2,0.8]])
p_wheeze =  Prob(Wheezing,[Bronchitis],[[0.999,0.001],[0.4,0.6]])

simple_diagnosis = BeliefNetwork("Simple Diagnosis",
                  {Influenza, Smokes, SoreThroat, HasFever, Bronchitis,
                      Coughing, Wheezing},
                  {p_infl, p_smokes, p_sth, p_fever, p_bronc, p_cough,
                      p_wheeze})
```

Sprinkler Example

The third belief network is the sprinkler example from Pearl [2009]. The output of `bn_sprinkler.show()` is shown in Figure 9.4 of this document.

probExamples.py — (continued)

```
Season = Variable("Season", ["dry_season","wet_season"],
    position=(0.5,0.9))
Sprinkler = Variable("Sprinkler", ["on","off"], position=(0.9,0.6))
Rained = Variable("Rained", boolean, position=(0.1,0.6))
Grass_wet = Variable("Grass wet", boolean, position=(0.5,0.3))
Grass_shiny = Variable("Grass shiny", boolean, position=(0.1,0))
Shoes_wet = Variable("Shoes wet", boolean, position=(0.9,0))

f_season = Prob(Season,[],{'dry_season':0.5, 'wet_season':0.5})
f_sprinkler = Prob(Sprinkler,[Season],{'dry_season':{'on':0.4,'off':0.6},
                                       'wet_season':{'on':0.01,'off':0.99}})
f_rained = Prob(Rained,[Season],{'dry_season':[0.9,0.1], 'wet_season':
    [0.2,0.8]})
f_wet = Prob(Grass_wet,[Sprinkler,Rained], {'on': [[0.1,0.9],[0.01,0.99]],
                                            'off':[[0.99,0.01],[0.3,0.7]]})
f_shiny = Prob(Grass_shiny, [Grass_wet], [[0.95,0.05], [0.3,0.7]])
f_shoes = Prob(Shoes_wet, [Grass_wet], [[0.98,0.02], [0.35,0.65]])
```

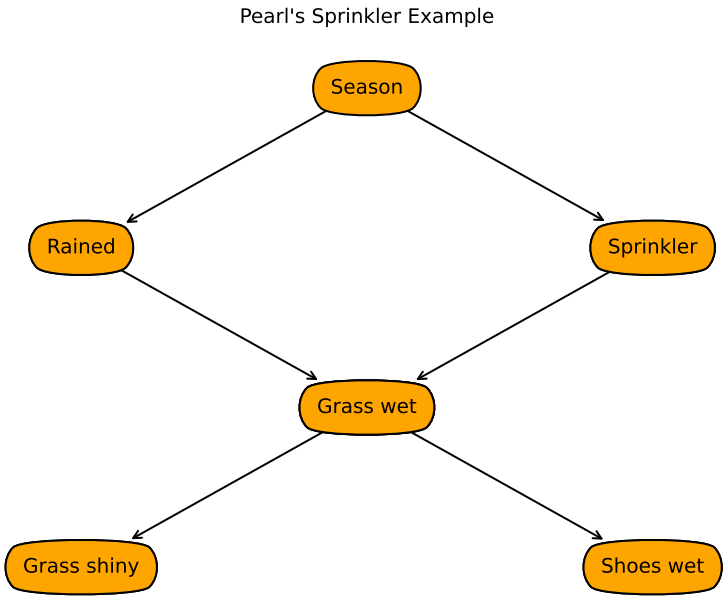


Figure 9.4: The sprinkler belief network

```
74
75 bn_sprinkler = BeliefNetwork("Pearl's Sprinkler Example",
76                         {Season, Sprinkler, Rained, Grass_wet, Grass_shiny,
                               Shoes_wet},
77                         {f_season, f_sprinkler, f_rained, f_wet, f_shiny,
                               f_shoes})
```

Bipartite Diagnostic Model with Noisy-or

The belief network `bn_no1` is a bipartite diagnostic model, with independent diseases, and the symptoms depend on the diseases, where the CPDs are defined using noisy-or. Bipartite means it is in two parts; the diseases are only connected to the symptoms and the symptoms are only connected to the diseases. The output of `bn_no1.show()` is shown in Figure 9.5 of this document.

probExamples.py — (continued)

```
79 #### Bipartite Diagnostic Network ###
80 Cough = Variable("Cough", boolean, (0.1,0.1))
81 Fever = Variable("Fever", boolean, (0.5,0.1))
82 Sneeze = Variable("Sneeze", boolean, (0.9,0.1))
83 Cold = Variable("Cold",boolean, (0.1,0.9))
84 Flu = Variable("Flu",boolean, (0.5,0.9))
```

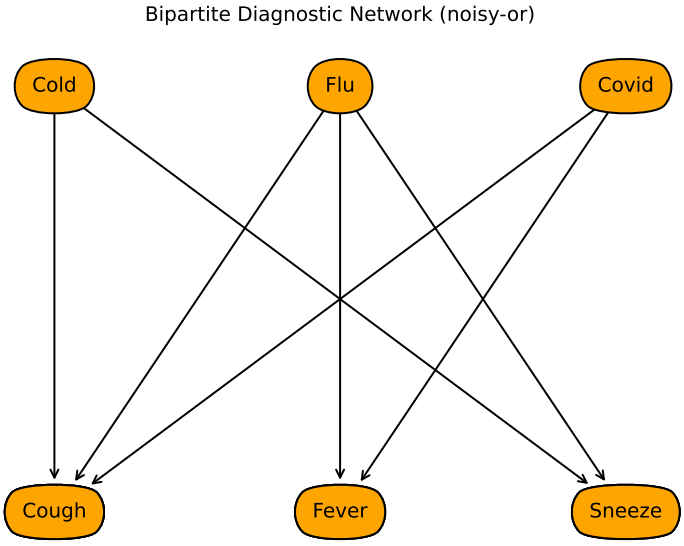


Figure 9.5: A bipartite diagnostic network

```
Covid = Variable("Covid",boolean, (0.9,0.9))

p_cold_no = Prob(Cold,[],[0.9,0.1])
p_flu_no = Prob(Flu,[],[0.95,0.05])
p_covid_no = Prob(Covid,[],[0.99,0.01])

p_cough_no = NoisyOR(Cough, [Cold,Flu,Covid], [0.1, 0.3, 0.2, 0.7])
p_fever_no = NoisyOR(Fever, [   Flu,Covid], [0.01,     0.6,  0.7])
p_sneeze_no = NoisyOR(Sneeze, [Cold,Flu  ], [0.05, 0.5,  0.2   ])

bn_no1 = BeliefNetwork("Bipartite Diagnostic Network (noisy-or)",
                       {Cough, Fever, Sneeze, Cold, Flu, Covid},
                        {p_cold_no, p_flu_no, p_covid_no, p_cough_no,
                            p_fever_no, p_sneeze_no})

# to see the conditional probability of Noisy-or do:
# print(p_cough_no.to_table())

# example from box "Noisy-or compared to logistic regression"
# X = Variable("X",boolean)
# w0 = 0.01
# print(NoisyOR(X,[A,B,C,D],[w0, 1-(1-0.05)/(1-w0), 1-(1-0.1)/(1-w0),
    1-(1-0.2)/(1-w0), 1-(1-0.2)/(1-w0), ]).to_table(given={X:True}))
```

Bipartite Diagnostic Model with Logistic Regression

The belief network bn_lr1 is a bipartite diagnostic model, with independent diseases, and the symptoms depend on the diseases, where the CPDs are defined using logistic regression. It has the same graphical structure as the previous example (see Figure 9.5). This has the (approximately) the same conditional probabilities as the previous example when zero or one diseases are present. Note that $sigmoid(-2.2) \approx 0.1$

probExamples.py — (continued)

```
107
108 p_cold_lr = Prob(Cold,[],[0.9,0.1])
109 p_flu_lr = Prob(Flu,[],[0.95,0.05])
110 p_covid_lr = Prob(Covid,[],[0.99,0.01])
111
112 p_cough_lr = LogisticRegression(Cough, [Cold,Flu,Covid], [-2.2, 1.67,
        1.26, 3.19])
113 p_fever_lr = LogisticRegression(Fever, [  Flu,Covid], [-4.6,      5.02,
        5.46])
114 p_sneeze_lr = LogisticRegression(Sneeze, [Cold,Flu ], [-2.94, 3.04, 1.79
        ])
115
116 bn_lr1 = BeliefNetwork("Bipartite Diagnostic Network - logistic
        regression",
117                        {Cough, Fever, Sneeze, Cold, Flu, Covid},
118                         {p_cold_lr, p_flu_lr, p_covid_lr, p_cough_lr,
                                p_fever_lr, p_sneeze_lr})
119
120 # to see the conditional probability of Noisy-or do:
121 #print(p_cough_lr.to_table())
122
123 # example from box "Noisy-or compared to logistic regression"
124 # from learnLinear import sigmoid, logit
125 # w0=logit(0.01)
126 # X = Variable("X",boolean)
127 # print(LogisticRegression(X,[A,B,C,D],[w0, logit(0.05)-w0, logit(0.1)-w0,
        logit(0.2)-w0, logit(0.2)-w0]).to_table(given={X:True}))
128 # try to predict what would happen (and then test) if we had
129 # w0=logit(0.01)
```

9.5 Inference Methods

Each of the inference methods implements the query method that computes the posterior probability of a variable given a dictionary of $\{variable : value\}$ observations. The methods are Displayable because they implement the *display* method which is text-based unless overridden.

probGraphicalModels.py — (continued)

```
from display import Displayable

class InferenceMethod(Displayable):
    """The abstract class of graphical model inference methods"""
    method_name = "unnamed" # each method should have a method name

    def __init__(self,gm=None):
        self.gm = gm

    def query(self, qvar, obs={}):
        """returns a {value:prob} dictionary for the query variable"""
        raise NotImplementedError("InferenceMethod query") # abstract method
```

We use bn_4ch as the test case, in particular $P(B \mid D = true)$. This needs an error threshold, particularly for the approximate methods, where the default threshold is much too accurate.

probGraphicalModels.py — (continued)

```
    def testIM(self, threshold=0.0000000001):
        solver = self(bn_4ch)
        res = solver.query(B,{D:True})
        correct_answer = 0.429632380245
        assert correct_answer-threshold < res[True] <
            correct_answer+threshold, \
               f"value {res[True]} not in desired range for
                   {self.method_name}"
        print(f"Unit test passed for {self.method_name}.")
```

9.5.1 Showing Posterior Distributions

The show_post method draws the posterior distribution of all variables. Figure 9.6 shows the result of bn_reportRC.show_post({Report:True}) when run after loading probRC.py (see below).

probGraphicalModels.py — (continued)

```
    def show_post(self, obs={}, num_format="{:.3f}", fontsize=10,
        facecolor='orange'):
        """draws the graphical model conditioned on observations obs
           num_format is number format (allows for more or less precision)
           fontsize gives size of the text
           facecolor gives the color of the nodes
        """
        plt.ion()  # interactive
        fig, ax = plt.subplots()
        ax.set_axis_off()
        ax.set_title(self.gm.title+" observed: "+str(obs),
            fontsize=fontsize)
        bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
            facecolor=facecolor)
        vartext = {} # variable:text dictionary
```

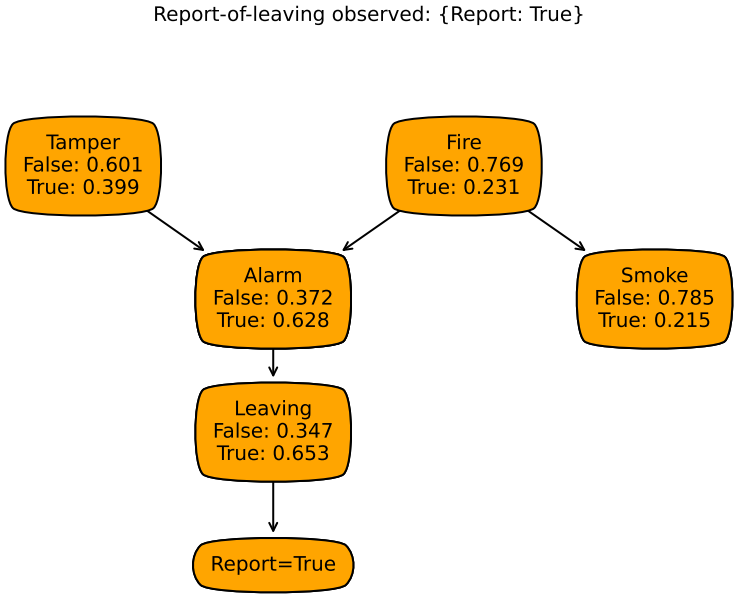


Figure 9.6: The report-of-leaving belief network with posterior distributions

```
        for var in self.gm.variables: #reversed(self.gm.topological_sort()):
            if var in obs:
                text = var.name + "=" + str(obs[var])
            else:
                distn = self.query(var, obs=obs)

                text = var.name + "\n" + "\n".join(str(d)+":
                    "+num_format.format(v) for (d,v) in distn.items())
            vartext[var] = text
            # Draw arcs
            for par in self.gm.var2parents[var]:
                    ax.annotate(text, par.position, xytext=var.position,
                                arrowprops={'arrowstyle':'<-'},bbox=bbox,
                                ha='center', va='center',
                                    fontsize=fontsize)
        for var in self.gm.variables:
            x,y = var.position
            ax.text(x,y,vartext[var], bbox=bbox, ha='center', va='center',
                fontsize=fontsize)
```

9.6 Naive Search

An instance of a *ProbSearch* object takes in a graphical model. The query method uses naive search to compute the probability of a query variable given observations on other variables. See Figure 9.9 of Poole and Mackworth [2023].

probRC.py — Search-based Inference for Graphical Models

```
import math
from probGraphicalModels import GraphicalModel, InferenceMethod
from probFactors import Factor

class ProbSearch(InferenceMethod):
    """The class that queries graphical models using search

    gm is graphical model to query
    """
    method_name = "naive search"

    def __init__(self,gm=None):
        InferenceMethod.__init__(self, gm)
        ## self.max_display_level = 3

    def query(self, qvar, obs={}, split_order=None):
        """computes P(qvar | obs) where
        qvar is the query variable
        obs is a variable:value dictionary
        split_order is a list of the non-observed non-query variables in gm
        """
        if qvar in obs:
            return {val:(1 if val == obs[qvar] else 0)
                        for val in qvar.domain}
        else:
           if split_order == None:
                split_order = [v for v in self.gm.variables
                               if (v not in obs) and v != qvar]
           unnorm = [self.prob_search({qvar:val}|obs, self.gm.factors,
                                      split_order)
                     for val in qvar.domain]
           p_obs = sum(unnorm)
           return {val:pr/p_obs for val,pr in zip(qvar.domain, unnorm)}
```

The following is the naive search-based algorithm. It is exponential in the number of variables, so is not very useful. However, it is simple, and helpful to understand before looking at the more complicated algorithm used in the subclass.

probRC.py — (continued)

```
    def prob_search(self, context, factors, split_order):
        """simple search algorithm
        context: a variable:value dictionary
        factors: a set of factors
```

```
48          split_order: list of variables not assigned in context
49          returns sum over variable assignments to variables in split order
                of product of factors """
50          self.display(2,"calling prob_search,",(context,factors,split_order))
51          if not factors:
52              return 1
53          elif to_eval := {fac for fac in factors
54                           if fac.can_evaluate(context)}:
55              # evaluate factors when all variables are assigned
56              self.display(3,"prob_search evaluating factors",to_eval)
57              val = math.prod(fac.get_value(context) for fac in to_eval)
58              return val * self.prob_search(context, factors-to_eval,
                    split_order)
59          else:
60              total = 0
61              var = split_order[0]
62              self.display(3, "prob_search branching on", var)
63              for val in var.domain:
64                  total += self.prob_search({var:val}|context, factors,
                        split_order[1:])
65              self.display(3, "prob_search branching on", var,"returning",
                    total)
66              return total
```

9.7 Recursive Conditioning

The **recursive conditioning (RC)** algorithm adds forgetting and caching and recognizing disconnected components to the naive search. We do this by adding a cache and redefining the recursive search algorithm. It inherits the query method. See Figure 9.12 of Poole and Mackworth [2023].

The cache is initialized with the empty context and empty factors has probability 1. This means that checking the cache can act as the base case when the context is empty.

probRC.py — (continued)

```
68  class ProbRC(ProbSearch):
69      method_name = "recursive conditioning"
70
71      def __init__(self,gm=None):
72          self.cache = {(frozenset(), frozenset()):1}
73          ProbSearch.__init__(self,gm)
74
75      def prob_search(self, context, factors, split_order):
76          """ returns sum_{split_order} prod_{factors} given assignment in
                context
77          context is a variable:value dictionary
78          factors is a set of factors
79          split_order: list of variables in factors that are not in context
```

```
        """
        self.display(3,"calling rc,",(context,factors))
        ce = (frozenset(context.items()), frozenset(factors)) # key for the
            cache entry
        if ce in self.cache:
            self.display(3,"rc cache lookup",(context,factors))
            return self.cache[ce]
        elif vars_not_in_factors := {var for var in context
                                     if not any(var in fac.variables
                                                for fac in factors)}:
            # forget variables not in any factor
            self.display(3,"rc forgetting variables", vars_not_in_factors)
            return self.prob_search({key:val for (key,val) in
                context.items()
                            if key not in vars_not_in_factors},
                        factors, split_order)
        elif to_eval := {fac for fac in factors
                         if fac.can_evaluate(context)}:
            # evaluate factors when all variables are assigned
            self.display(3,"rc evaluating factors",to_eval)
            val = math.prod(fac.get_value(context) for fac in to_eval)
            if val == 0:
                return 0
            else:
             return val * self.prob_search(context,
                                     {fac for fac in factors
                                              if fac not in to_eval},
                                     split_order)
        elif len(comp := connected_components(context, factors,
            split_order)) > 1:
            # there are disconnected components
            self.display(3,"splitting into connected components",comp,"in
                context",context)
            return(math.prod(self.prob_search(context,f,eo) for (f,eo) in
                comp))
        else:
            assert split_order, "split_order should not be empty to get
                here"
            total = 0
            var = split_order[0]
            self.display(3, "rc branching on", var)
            for val in var.domain:
                total += self.prob_search({var:val}|context, factors,
                    split_order[1:])
            self.cache[ce] = total
            self.display(2, "rc branching on", var,"returning", total)
            return total
```

`connected_components` returns a list of connected components, where a connected component is a set of factors and a set of variables, where the graph that connects variables and factors that involve them is connected. The connected

components are built one at a time; with a current connected component. At all times `factors` is partitioned into 3 disjoint sets:

- `component_factors` containing factors in the current connected component where all factors that share a variable are already in the component
- `factors_to_check` containing factors in the current connected component where potentially some factors that share a variable are not in the component; these need to be checked
- `other_factors` the other factors that are not (yet) in the connected component

probRC.py — (continued)

```
121 def connected_components(context, factors, split_order):
122     """returns a list of (f,e) where f is a subset of factors and e is a
            subset of split_order
123     such that each element shares the same variables that are disjoint from
            other elements.
124     """
125     other_factors = set(factors) #copies factors
126     factors_to_check = {other_factors.pop()} # factors in connected
            component still to be checked
127     component_factors = set() # factors in first connected component
            already checked
128     component_variables = set() # variables in first connected component
129     while factors_to_check:
130         next_fac = factors_to_check.pop()
131         component_factors.add(next_fac)
132         new_vars = set(next_fac.variables) - component_variables -
                context.keys()
133         component_variables |= new_vars
134         for var in new_vars:
135             factors_to_check |= {f for f in other_factors
136                                  if var in f.variables}
137             other_factors -= factors_to_check # set difference
138     if  other_factors:
139         return ( [(component_factors,[e for e in split_order
140                                       if e in component_variables])]
141                 + connected_components(context, other_factors,
142                                        [e for e in split_order
143                                           if e not in component_variables]) )
144     else:
145         return [(component_factors, split_order)]
```

Testing:

probRC.py — (continued)

```
147 from probGraphicalModels import bn_4ch, A,B,C,D,f_a,f_b,f_c,f_d
148 bn_4chv = ProbRC(bn_4ch)
```

```
## bn_4chv.query(A,{})
## bn_4chv.query(D,{})
## InferenceMethod.max_display_level = 3 # show more detail in displaying
## InferenceMethod.max_display_level = 1 # show less detail in displaying
## bn_4chv.query(A,{D:True},[C,B])
## bn_4chv.query(B,{A:True,D:False})

from probExamples import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
bn_reportRC = ProbRC(bn_report) # answers queries using recursive
    conditioning
## bn_reportRC.query(Tamper,{})
## InferenceMethod.max_display_level = 0 # show no detail in displaying
## bn_reportRC.query(Leaving,{})
## bn_reportRC.query(Tamper,{},
    split_order=[Smoke,Fire,Alarm,Leaving,Report])
## bn_reportRC.query(Tamper,{Report:True})
## bn_reportRC.query(Tamper,{Report:True,Smoke:False})

## To display resulting posteriors try:
# bn_reportRC.show_post({})
# bn_reportRC.show_post({Smoke:False})
# bn_reportRC.show_post({Report:True})
# bn_reportRC.show_post({Report:True, Smoke:False})

## Note what happens to the cache when these are called in turn:
## bn_reportRC.query(Tamper,{Report:True},
    split_order=[Smoke,Fire,Alarm,Leaving])
## bn_reportRC.query(Smoke,{Report:True},
    split_order=[Tamper,Fire,Alarm,Leaving])

from probExamples import bn_sprinkler, Season, Sprinkler, Rained,
    Grass_wet, Grass_shiny, Shoes_wet
bn_sprinklerv = ProbRC(bn_sprinkler)
## bn_sprinklerv.query(Shoes_wet,{})
## bn_sprinklerv.query(Shoes_wet,{Rained:True})
## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:True})
## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:False,Rained:True})

from probExamples import bn_no1, bn_lr1, Cough, Fever, Sneeze, Cold, Flu,
    Covid
bn_no1v = ProbRC(bn_no1)
bn_lr1v = ProbRC(bn_lr1)
## bn_no1v.query(Flu, {Fever:1, Sneeze:1})
## bn_lr1v.query(Flu, {Fever:1, Sneeze:1})
## bn_lr1v.query(Cough,{})
## bn_lr1v.query(Cold,{Cough:1,Sneeze:0,Fever:1})
## bn_lr1v.query(Flu,{Cough:0,Sneeze:1,Fever:1})
## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1})
## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:0})
## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:1})
```

```
193
194 if __name__ == "__main__":
195     InferenceMethod.testIM(ProbSearch)
196     InferenceMethod.testIM(ProbRC)
```

The following example uses the decision tree representation of Section 9.3.4 (page 210).

probRC.py — (continued)

```
198 from probFactors import Prob, action, rain, full, wet, p_wet
199 from probGraphicalModels import BeliefNetwork
200 p_action = Prob(action,[],{'go_out':0.3, 'get_coffee':0.7})
201 p_rain = Prob(rain,[],[0.4,0.6])
202 p_full = Prob(full,[],[0.1,0.9])
203
204 wetBN = BeliefNetwork("Wet (decision tree CPD)", {action, rain, full, wet},
205                       {p_action, p_rain, p_full, p_wet})
206 wetRC = ProbRC(wetBN)
207 # wetRC.query(wet, {action:'go_out', rain:True})
208 # wetRC.show_post({action:'go_out', rain:True})
209 # wetRC.show_post({action:'go_out', wet:True})
```

Exercise 9.1 Does recursive conditioning split on variable `full` for the query commented out above? Does it need to? Fix the code so that decision tree representations of conditional probabilities can be evaluated as soon as possible.

Exercise 9.2 This code adds to the cache only after splitting. Implement a variant that caches after forgetting. (What can the cache start with?) Which version works better? Compare some measure of the search tree and the space used. Try other alternatives of what to cache; which method works best?

9.8 Variable Elimination

An instance of a *VE* object takes in a graphical model. The query method uses variable elimination to compute the probability of a variable given observations on some other variables.

probVE.py — Variable Elimination for Graphical Models

```
11 from probFactors import Factor, FactorObserved, FactorSum, factor_times
12 from probGraphicalModels import GraphicalModel, InferenceMethod
13
14 class VE(InferenceMethod):
15     """The class that queries Graphical Models using variable elimination.
16
17     gm is graphical model to query
18     """
19     method_name = "variable elimination"
20
21     def __init__(self,gm=None):
22         InferenceMethod.__init__(self, gm)
```

```
23
24      def query(self,var,obs={},elim_order=None):
25          """computes P(var|obs) where
26          var is a variable
27          obs is a {variable:value} dictionary"""
28          if var in obs:
29              return {var:1 if val == obs[var] else 0 for val in var.domain}
30          else:
31              if elim_order == None:
32                  elim_order = self.gm.variables
33              projFactors = [self.project_observations(fact,obs)
34                             for fact in self.gm.factors]
35              for v in elim_order:
36                  if v != var and v not in obs:
37                      projFactors = self.eliminate_var(projFactors,v)
38              unnorm = factor_times(var,projFactors)
39              p_obs=sum(unnorm)
40              self.display(1,"Unnormalized probs:",unnorm,"Prob obs:",p_obs)
41              return {val:pr/p_obs for val,pr in zip(var.domain, unnorm)}
```

A *FactorObserved* is a factor that is the result of some observations on another factor. We don't store the values in a list; we just look them up as needed. The observations can include variables that are not in the list, but should have some intersection with the variables in the factor.

probFactors.py — (continued)

```
237 class FactorObserved(Factor):
238     def __init__(self,factor,obs):
239         Factor.__init__(self, [v for v in factor.variables if v not in obs])
240         self.observed = obs
241         self.orig_factor = factor
242
243     def get_value(self,assignment):
244         return self.orig_factor.get_value(assignment|self.observed)
```

A *FactorSum* is a factor that is the result of summing out a variable from the product of other factors. I.e., it constructs a representation of:

$$\sum_{var} \prod_{f \in factors} f(var).$$

We store the values in a list in a lazy manner; if they are already computed, we used the stored values. If they are not already computed we can compute and store them.

probFactors.py — (continued)

```
246 class FactorSum(Factor):
247     def __init__(self,var,factors):
248         self.var_summed_out = var
249         self.factors = factors
250         vars = list({v for fac in factors
```

```
                    for v in fac.variables if v is not var})
        #for fac in factors:
        #    for v in fac.variables:
        #        if v is not var and v not in vars:
        #            vars.append(v)
        Factor.__init__(self,vars)
        self.values = {}

    def get_value(self,assignment):
        """lazy implementation: if not saved, compute it. Return saved
            value"""
        asst = frozenset(assignment.items())
        if asst in self.values:
            return self.values[asst]
        else:
            total = 0
            new_asst = assignment.copy()
            for val in self.var_summed_out.domain:
                new_asst[self.var_summed_out] = val
                total += math.prod(fac.get_value(new_asst) for fac in
                    self.factors)
            self.values[asst] = total
            return total
```

The method *factor_times* multiplies a set of factors that are all factors on the same variable (or on no variables). This is the last step in variable elimination before normalizing. It returns an array giving the product for each value of *variable*.

probFactors.py — (continued)

```
def factor_times(variable, factors):
    """when factors are factors just on variable (or on no variables)"""
    prods = []
    facs = [f for f in factors if variable in f.variables]
    for val in variable.domain:
        ast = {variable:val}
        prods.append(math.prod(f.get_value(ast) for f in facs))
    return prods
```

To project observations onto a factor, for each variable that is observed in the factor, we construct a new factor that is the factor projected onto that variable. *Factor_observed* creates a new factor that is the result is assigning a value to a single variable.

probVE.py — (continued)

```
    def project_observations(self,factor,obs):
        """Returns the resulting factor after observing obs

        obs is a dictionary of {variable:value} pairs.
        """
        if any((var in obs) for var in factor.variables):
```

```
            # a variable in factor is observed
            return FactorObserved(factor,obs)
        else:
            return factor

    def eliminate_var(self,factors,var):
        """Eliminate a variable var from a list of factors.
        Returns a new set of factors that has var summed out.
        """
        self.display(2,"eliminating ",str(var))
        contains_var = []
        not_contains_var = []
        for fac in factors:
            if var in fac.variables:
                contains_var.append(fac)
            else:
                not_contains_var.append(fac)
        if contains_var == []:
            return factors
        else:
            newFactor = FactorSum(var,contains_var)
            self.display(2,"Multiplying:",[str(f) for f in contains_var])
            self.display(2,"Creating factor:", newFactor)
            self.display(3, newFactor.to_table()) # factor in detail
            not_contains_var.append(newFactor)
            return not_contains_var

from probGraphicalModels import bn_4ch, A,B,C,D
bn_4chv = VE(bn_4ch)
## bn_4chv.query(A,{})
## bn_4chv.query(D,{})
## InferenceMethod.max_display_level = 3 # show more detail in displaying
## InferenceMethod.max_display_level = 1 # show less detail in displaying
## bn_4chv.query(A,{D:True})
## bn_4chv.query(B,{A:True,D:False})

from probExamples import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
bn_reportv = VE(bn_report) # answers queries using variable elimination
## bn_reportv.query(Tamper,{})
## InferenceMethod.max_display_level = 0 # show no detail in displaying
## bn_reportv.query(Leaving,{})
## bn_reportv.query(Tamper,{},elim_order=[Smoke,Report,Leaving,Alarm,Fire])
## bn_reportv.query(Tamper,{Report:True})
## bn_reportv.query(Tamper,{Report:True,Smoke:False})

from probExamples import bn_sprinkler, Season, Sprinkler, Rained,
    Grass_wet, Grass_shiny, Shoes_wet
bn_sprinklerv = VE(bn_sprinkler)
## bn_sprinklerv.query(Shoes_wet,{})
## bn_sprinklerv.query(Shoes_wet,{Rained:True})
```

```
## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:True})
## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:False,Rained:True})

from probExamples import bn_lr1, Cough, Fever, Sneeze, Cold, Flu, Covid
vediag = VE(bn_lr1)
## vediag.query(Cough,{})
## vediag.query(Cold,{Cough:1,Sneeze:0,Fever:1})
## vediag.query(Flu,{Cough:0,Sneeze:1,Fever:1})
## vediag.query(Covid,{Cough:1,Sneeze:0,Fever:1})
## vediag.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:0})
## vediag.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:1})

if __name__ == "__main__":
    InferenceMethod.testIM(VE)
```

9.9 Stochastic Simulation

9.9.1 Sampling from a discrete distribution

The method *sample_one* generates a single sample from a (possibly unnormalized) distribution. *dist* is a $\{value : weight\}$ dictionary, where $weight \geq 0$. This returns a value with probability in proportion to its weight.

probStochSim.py — Probabilistic inference using stochastic simulation

```
import random
from probGraphicalModels import InferenceMethod

def sample_one(dist):
    """returns the index of a single sample from normalized distribution
        dist."""
    rand = random.random()*sum(dist.values())
    cum = 0    # cumulative weights
    for v in dist:
        cum += dist[v]
        if cum > rand:
            return v
```

If we want to generate multiple samples, repeatedly calling *sample_one* may not be efficient. If we want to generate multiple samples, and the distribution is over m values, it searches through the m values of the distribution for each sample.

The method *sample_multiple* generates multiple samples from a distribution defined by *dist*, where *dist* is a $\{value : weight\}$ dictionary, where $weight \geq 0$ and the weights are not all zero. This returns a list of values, of length *num_samples*, where each sample is selected with a probability proportional to its weight.

The method generates all of the random numbers, sorts them, and then goes through the distribution once, saving the selected samples.

probStochSim.py — (continued)

```
def sample_multiple(dist, num_samples):
    """returns a list of num_samples values selected using distribution
        dist.
    dist is a {value:weight} dictionary that does not need to be normalized
    """
    total = sum(dist.values())
    rands = sorted(random.random()*total for i in range(num_samples))
    result = []
    dist_items = list(dist.items())
    cum = dist_items[0][1] # cumulative sum
    index = 0
    for r in rands:
        while r>cum:
            index += 1
            cum += dist_items[index][1]
        result.append(dist_items[index][0])
    return result
```

Exercise 9.3

What is the time and space complexity of the following 4 methods to generate n samples, where m is the length of *dist*:

(a) n calls to *sample_one*

(b) *sample_multiple*

(c) Create the cumulative distribution (choose how this is represented) and, for each random number, do a binary search to determine the sample associated with the random number.

(d) Choose a random number in the range $[i/n, (i+1)/n)$ for each $i \in range(n)$, where n is the number of samples. Use these as the random numbers to select the particles. (Does this give random samples?)

For each method suggest when it might be the best method.

The *test_sampling* method can be used to generate the statistics from a number of samples. It is useful to see the variability as a function of the number of samples. Try it for a few samples and also for many samples.

probStochSim.py — (continued)

```
def test_sampling(dist, num_samples):
    """Given a distribution, dist, draw num_samples samples
    and return the resulting counts
    """
    result = {v:0 for v in dist}
    for v in sample_multiple(dist, num_samples):
        result[v] += 1
    return result

# try the following queries a number of times each:
# test_sampling({1:1,2:2,3:3,4:4}, 100)
# test_sampling({1:1,2:2,3:3,4:4}, 100000)
```

9.9.2 Sampling Methods for Belief Network Inference

A *SamplingInferenceMethod* is an *InferenceMethod,* but the query method also takes arguments for the number of samples and the sample-order (which is an ordering of factors). The first methods assume a belief network (and not an undirected graphical model).

probStochSim.py — (continued)

```
53 class SamplingInferenceMethod(InferenceMethod):
54     """The abstract class of sampling-based belief network inference
           methods"""
55
56     def __init__(self,gm=None):
57         InferenceMethod.__init__(self, gm)
58
59     def query(self,qvar,obs={},number_samples=1000,sample_order=None):
60         raise NotImplementedError("SamplingInferenceMethod query") #
               abstract
```

9.9.3 Rejection Sampling

probStochSim.py — (continued)

```
62 class RejectionSampling(SamplingInferenceMethod):
63     """The class that queries Graphical Models using Rejection Sampling.
64
65     gm is a belief network to query
66     """
67     method_name = "rejection sampling"
68
69     def __init__(self, gm=None):
70         SamplingInferenceMethod.__init__(self, gm)
71
72     def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
73         """computes P(qvar | obs) where
74         qvar is a variable.
75         obs is a {variable:value} dictionary.
76         sample_order is a list of variables where the parents
77           come before the variable.
78         """
79         if sample_order is None:
80             sample_order = self.gm.topological_sort()
81         self.display(2,*sample_order,sep="\t")
82         counts = {val:0 for val in qvar.domain}
83         for i in range(number_samples):
84             rejected = False
85             sample = {}
86             for nvar in sample_order:
87                 fac = self.gm.var2cpt[nvar]  #factor with nvar as child
```

```
88              val = sample_one({v:fac.get_value({**sample, nvar:v}) for v
                    in nvar.domain})
89              self.display(2,val,end="\t")
90              if nvar in obs and obs[nvar] != val:
91                  rejected = True
92                  self.display(2,"Rejected")
93                  break
94              sample[nvar] = val
95          if not rejected:
96              counts[sample[qvar]] += 1
97              self.display(2,"Accepted")
98      tot = sum(counts.values())
99      # As well as the distribution we also include raw counts
100     dist = {c:v/tot if tot>0 else 1/len(qvar.domain) for (c,v) in
            counts.items()}
101     dist["raw_counts"] = counts
102     return dist
```

9.9.4 Likelihood Weighting

Likelihood weighting includes a weight for each sample. Instead of rejecting samples based on observations, likelihood weighting changes the weights of the sample in proportion with the probability of the observation. The weight then becomes the probability that the variable would have been rejected.

probStochSim.py — (continued)

```
104 class LikelihoodWeighting(SamplingInferenceMethod):
105     """The class that queries Graphical Models using Likelihood weighting.
106
107     gm is a belief network to query
108     """
109     method_name = "likelihood weighting"
110
111     def __init__(self, gm=None):
112         SamplingInferenceMethod.__init__(self, gm)
113
114     def query(self,qvar,obs={},number_samples=1000,sample_order=None):
115         """computes P(qvar | obs) where
116         qvar is a variable.
117         obs is a {variable:value} dictionary.
118         sample_order is a list of factors where factors defining the parents
119           come before the factors for the child.
120         """
121         if sample_order is None:
122             sample_order = self.gm.topological_sort()
123         self.display(2,*[v for v in sample_order
124                          if v not in obs],sep="\t")
125         counts = {val:0 for val in qvar.domain}
126         for i in range(number_samples):
127             sample = {}
128             weight = 1.0
```

```
            for nvar in sample_order:
                fac = self.gm.var2cpt[nvar]
                if nvar in obs:
                    sample[nvar] = obs[nvar]
                    weight *= fac.get_value(sample)
                else:
                    val = sample_one({v:fac.get_value({**sample,nvar:v}) for
                        v in nvar.domain})
                    self.display(2,val,end="\t")
                    sample[nvar] = val
            counts[sample[qvar]] += weight
            self.display(2,weight)
        tot = sum(counts.values())
        # as well as the distribution we also include the raw counts
        dist = {c:v/tot for (c,v) in counts.items()}
        dist["raw_counts"] = counts
        return dist
```

Exercise 9.4 Change this algorithm so that it does **importance sampling** using a proposal distribution that may be different from the prior. It needs *sample_one* using a different distribution and then adjust the weight of the current sample. For testing, use a proposal distribution that only differs from the prior for a subset of the variables. For which variables does the different proposal distribution make the most difference?

9.9.5 Particle Filtering

In this implementation, a particle is a $\{variable : value\}$ dictionary. Because adding a new value to dictionary involves a side effect, the dictionaries are copied during resampling.

probStochSim.py — (continued)

```
class ParticleFiltering(SamplingInferenceMethod):
    """The class that queries Graphical Models using Particle Filtering.

    gm is a belief network to query
    """
    method_name = "particle filtering"

    def __init__(self, gm=None):
        SamplingInferenceMethod.__init__(self, gm)

    def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
        """computes P(qvar | obs) where
        qvar is a variable.
        obs is a {variable:value} dictionary.
        sample_order is a list of factors where factors defining the parents
          come before the factors for the child.
        """
        if sample_order is None:
```

```
164             sample_order = self.gm.topological_sort()
165         self.display(2,*[v for v in sample_order
166                          if v not in obs],sep="\t")
167         particles = [{} for i in range(number_samples)]
168         for nvar in sample_order:
169             fac = self.gm.var2cpt[nvar]
170             if nvar in obs:
171                 weights = [fac.get_value({**part, nvar:obs[nvar]})
172                            for part in particles]
173                 particles = [{**p, nvar:obs[nvar]}
174                              for p in resample(particles, weights,
                                     number_samples)]
175             else:
176                 for part in particles:
177                     part[nvar] = sample_one({v:fac.get_value({**part,
                            nvar:v})
178                                              for v in nvar.domain})
179                 self.display(2,part[nvar],end="\t")
180         counts = {val:0 for val in qvar.domain}
181         for part in particles:
182             counts[part[qvar]] += 1
183         tot = sum(counts.values())
184         # as well as the distribution we also include the raw counts
185         dist = {c:v/tot for (c,v) in counts.items()}
186         dist["raw_counts"] = counts
187         return dist
```

Resampling

Resample is based on *sample_multiple* but works with an array of particles. (Aside: Python doesn't let us use *sample_multiple* directly as it uses a dictionary and particles, represented as dictionaries can't be the key of dictionaries).

probStochSim.py — (continued)

```
189 def resample(particles, weights, num_samples):
190     """returns num_samples copies of particles resampled according to
            weights.
191     particles is a list of particles
192     weights is a list of positive numbers, of same length as particles
193     num_samples is n integer
194     """
195     total = sum(weights)
196     rands = sorted(random.random()*total for i in range(num_samples))
197     result = []
198     cum = weights[0]   # cumulative sum
199     index = 0
200     for r in rands:
201         while r>cum:
202             index += 1
203             cum += weights[index]
```

```
204         result.append(particles[index])
205     return result
```

9.9.6 Examples

probStochSim.py — (continued)

```
207 from probGraphicalModels import bn_4ch, A,B,C,D
208 bn_4chr = RejectionSampling(bn_4ch)
209 bn_4chL = LikelihoodWeighting(bn_4ch)
210 ## InferenceMethod.max_display_level = 2 # detailed tracing for all
        inference methods
211 ## bn_4chr.query(A,{})
212 ## bn_4chr.query(C,{})
213 ## bn_4chr.query(A,{C:True})
214 ## bn_4chr.query(B,{A:True,C:False})
215
216 from probExamples import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
217 bn_reportr = RejectionSampling(bn_report) # answers queries using
        rejection sampling
218 bn_reportL = LikelihoodWeighting(bn_report) # answers queries using
        likelihood weighting
219 bn_reportp = ParticleFiltering(bn_report) # answers queries using particle
        filtering
220 ## bn_reportr.query(Tamper,{})
221 ## bn_reportr.query(Tamper,{})
222 ## bn_reportr.query(Tamper,{Report:True})
223 ## InferenceMethod.max_display_level = 0 # no detailed tracing for all
        inference methods
224 ## bn_reportr.query(Tamper,{Report:True},number_samples=100000)
225 ## bn_reportr.query(Tamper,{Report:True,Smoke:False})
226 ## bn_reportr.query(Tamper,{Report:True,Smoke:False},number_samples=100)
227
228 ## bn_reportL.query(Tamper,{Report:True,Smoke:False},number_samples=100)
229 ## bn_reportL.query(Tamper,{Report:True,Smoke:False},number_samples=100)
230
231 from probExamples import bn_sprinkler,Season, Sprinkler
232 from probExamples import Rained, Grass_wet, Grass_shiny, Shoes_wet
233 bn_sprinklerr = RejectionSampling(bn_sprinkler) # answers queries using
        rejection sampling
234 bn_sprinklerL = LikelihoodWeighting(bn_sprinkler) # answers queries using
        rejection sampling
235 bn_sprinklerp = ParticleFiltering(bn_sprinkler) # answers queries using
        particle filtering
236 #bn_sprinklerr.query(Shoes_wet,{Grass_shiny:True,Rained:True})
237 #bn_sprinklerL.query(Shoes_wet,{Grass_shiny:True,Rained:True})
238 #bn_sprinklerp.query(Shoes_wet,{Grass_shiny:True,Rained:True})
239
240 if __name__ == "__main__":
241     InferenceMethod.testIM(RejectionSampling, threshold=0.1)
```

```
    InferenceMethod.testIM(LikelihoodWeighting, threshold=0.1)
    InferenceMethod.testIM(ParticleFiltering, threshold=0.1)
```

9.9.7 Gibbs Sampling

The following implements **Gibbs sampling**, a form of **Markov Chain Monte Carlo** MCMC.

probStochSim.py — (continued)

```
#import random
#from probGraphicalModels import InferenceMethod

#from probStochSim import sample_one, SamplingInferenceMethod

class GibbsSampling(SamplingInferenceMethod):
    """The class that queries Graphical Models using Gibbs Sampling.

    bn is a graphical model (e.g., a belief network) to query
    """
    method_name = "Gibbs sampling"

    def __init__(self, gm=None):
        SamplingInferenceMethod.__init__(self, gm)
        self.gm = gm

    def query(self, qvar, obs={}, number_samples=1000, burn_in=100,
              sample_order=None):
        """computes P(qvar | obs) where
        qvar is a variable.
        obs is a {variable:value} dictionary.
        sample_order is a list of non-observed variables in order, or
        if sample_order None, an arbitrary ordering is used
        """
        counts = {val:0 for val in qvar.domain}
        if sample_order is not None:
            variables = sample_order
        else:
            variables = [v for v in self.gm.variables if v not in obs]
            random.shuffle(variables)
        var_to_factors = {v:set() for v in self.gm.variables}
        for fac in self.gm.factors:
            for var in fac.variables:
                var_to_factors[var].add(fac)
        sample = {var:random.choice(var.domain) for var in variables}
        self.display(3,"Sample:",sample)
        sample.update(obs)
        for i in range(burn_in + number_samples):
            for var in variables:
                # get unnormalized probability distribution of var given its
                    neighbors
                vardist = {val:1 for val in var.domain}
```

```
285                 for val in var.domain:
286                     sample[var] = val
287                     for fac in var_to_factors[var]: # Markov blanket
288                         vardist[val] *= fac.get_value(sample)
289                 sample[var] = sample_one(vardist)
290             if i >= burn_in:
291                 counts[sample[qvar]] +=1
292                 self.display(3,"     ",sample)
293         tot = sum(counts.values())
294         # as well as the computed distribution, we also include raw counts
295         dist = {c:v/tot for (c,v) in counts.items()}
296         dist["raw_counts"] = counts
297         self.display(2, f"Gibbs sampling P({qvar}|{obs}) = {dist}")
298         return dist
299 
300 #from probGraphicalModels import bn_4ch, A,B,C,D
301 bn_4chg = GibbsSampling(bn_4ch)
302 ## InferenceMethod.max_display_level = 2 # detailed tracing for all
        inference methods
303 bn_4chg.query(A,{})
304 ## bn_4chg.query(D,{})
305 ## bn_4chg.query(B,{D:True})
306 ## bn_4chg.query(B,{A:True,C:False})
307 
308 from probExamples import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
309 bn_reportg = GibbsSampling(bn_report)
310 ## bn_reportg.query(Tamper,{Report:True},number_samples=1000)
311 
312 if __name__ == "__main__":
313     InferenceMethod.testIM(GibbsSampling, threshold=0.1)
```

Exercise 9.5 Change the code so that it can have multiple query variables. Make the list of query variable be an input to the algorithm, so that the default value is the list of all non-observed variables.

Exercise 9.6 In this algorithm, explain where it computes the probability of a variable given its Markov blanket. Instead of returning the average of the samples for the query variable, it is possible to return the average estimate of the probability of the query variable given its Markov blanket. Does this converge to the same answer as the given code? Does it converge faster, slower, or the same?

9.9.8 Plotting Behavior of Stochastic Simulators

The stochastic simulation runs can give different answers each time they are run. For the algorithms that give the same answer in the limit as the number of samples approaches infinity (as do all of these algorithms), the algorithms can be compared by comparing the accuracy for multiple runs. Summary statistics like the variance may provide some information, but the assumptions behind the variance being appropriate (namely that the distribution is approximately

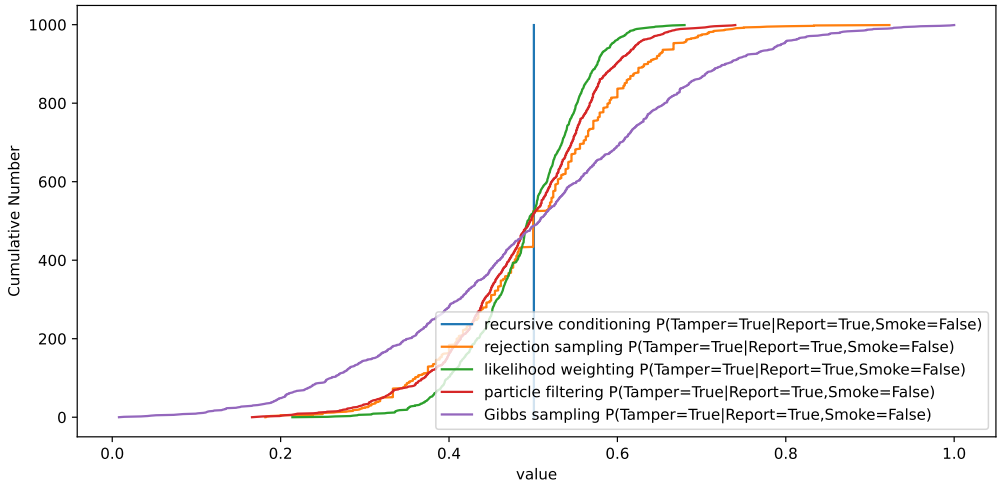


Figure 9.7: Cumulative distribution of the prediction of various models for $P(Tamper{=}True \mid report \wedge \neg smoke)$

Gaussian) may not hold for cases where the predictions are bounded and often skewed.

It is more appropriate to plot the distribution of predictions over multiple runs. The *plot_stats* method plots the prediction of a particular variable (or for the partition function) for a number of runs of the same algorithm. On the x-axis, is the prediction of the algorithm. On the y-axis is the number of runs with prediction less than or equal to the x value. Thus this is like a cumulative distribution over the predictions, but with counts on the y-axis.

Note that for runs where there are no samples that are consistent with the observations (as can happen with rejection sampling), the prediction of probability is 1.0 (as a convention for $0/0$).

That variable *what* contains the query variable, or if *what* is "*prob_ev*", the probability of evidence.

Figure 9.7 shows the distribution of various models. This figure is generated using the first `plot_mult` example below. Recursive conditioning gives the exact answer, and so is a vertical line. The others provide the cumulative prediction for 1000 runs for each method. This graph shows that for this graph and query, likelihood weighting is closest to the exact answer.

probStochSim.py — (continued)

```
315 import matplotlib.pyplot as plt
316
317 def plot_stats(method, qvar, qval, obs, number_runs=1000, **queryargs):
318     """Plots a cumulative distribution of the prediction of the model.
319     method is a InferenceMethod (that implements appropriate query(.))
320     plots P(qvar=qval | obs)
321     qvar is the query variable, qval is corresponding value
322     obs is the {variable:value} dictionary representing the observations
```

```
    number_iterations is the number of runs that are plotted
    **queryargs is the arguments to query (often number_samples for
        sampling methods)
    """
    plt.ion()
    # ax is global
    ax.set_xlabel("value")
    ax.set_ylabel("Cumulative Number")
    method.max_display_level, prev_mdl = 0, method.max_display_level #no
        display
    answers = [method.query(qvar,obs,**queryargs)
              for i in range(number_runs)]
    values = [ans[qval] for ans in answers]
    label = f"""{method.method_name}
        P({qvar}={qval}|{','.join(f'{var}={val}'
                                                  for (var,val) in
                                                      obs.items())})"""
    values.sort()
    ax.plot(values,range(number_runs),label=label)
    ax.legend() #loc="upper left")
    plt.show()
    method.max_display_level = prev_mdl # restore display level

if __name__ == "__main__":
    fig, ax = plt.subplots()

# Try:
# plot_stats(bn_reportr,Tamper,True,{Report:True,Smoke:True},
    number_samples=1000, number_runs=1000)
# plot_stats(bn_reportL,Tamper,True,{Report:True,Smoke:True},
    number_samples=1000, number_runs=1000)
# plot_stats(bn_reportp,Tamper,True,{Report:True,Smoke:True},
    number_samples=1000, number_runs=1000)
# plot_stats(bn_reportr,Tamper,True,{Report:True,Smoke:True},
    number_samples=100, number_runs=1000)
# plot_stats(bn_reportL,Tamper,True,{Report:True,Smoke:True},
    number_samples=100, number_runs=1000)
# plot_stats(bn_reportg,Tamper,True,{Report:True,Smoke:True},
    number_samples=1000, number_runs=1000)

def plot_mult(methods, example, qvar, qval, obs, number_samples=1000,
    number_runs=1000):
    for method in methods:
        solver = method(example)
        if isinstance(method,SamplingInferenceMethod):
            plot_stats(solver, qvar, qval, obs,
                number_samples=number_samples, number_runs=number_runs)
        else:
            plot_stats(solver, qvar, qval, obs, number_runs=number_runs)

```

```
from probRC import ProbRC
# Try following (but it takes a while..)
methods = [ProbRC, RejectionSampling, LikelihoodWeighting,
    ParticleFiltering, GibbsSampling]
#plot_mult(methods,bn_report,Tamper,True,{Report:True,Smoke:False},
    number_samples=100, number_runs=1000)
# plot_mult(methods,bn_report,Tamper,True,{Report:False,Smoke:True},
    number_samples=100, number_runs=1000)

# Sprinkler Example:
# plot_stats(bn_sprinklerr,Shoes_wet,True,{Grass_shiny:True,Rained:True},
    number_samples=1000)
# plot_stats(bn_sprinklerL,Shoes_wet,True,{Grass_shiny:True,Rained:True},
    number_samples=1000)
```

9.10 Hidden Markov Models

This code for hidden Markov models (HMMs) is independent of the graphical models code, to keep it simple. Section 9.11 gives code that models hidden Markov models, and more generally, dynamic belief networks, using the graphical models code.

This HMM code assumes there are multiple Boolean observation variables that depend on the current state and are independent of each other given the state.

probHMM.py — Hidden Markov Model

```
import random
from probStochSim import sample_one, sample_multiple

class HMM(object):
    def __init__(self, states, obsvars, pobs, trans, indist):
        """A hidden Markov model.
        states - set of states
        obsvars - set of observation variables
        pobs - probability of observations, pobs[i][s] is P(Obs_i=True |
            State=s)
        trans - transition probability - trans[i][j] gives P(State=j |
            State=i)
        indist - initial distribution - indist[s] is P(State_0 = s)
        """
        self.states = states
        self.obsvars = obsvars
        self.pobs = pobs
        self.trans = trans
        self.indist = indist
```

Consider the following example. Suppose you want to unobtrusively keep track of an animal in a triangular enclosure using sound. Suppose you have

3 microphones that provide unreliable (noisy) binary information at each time step. The animal is either close to one of the 3 points of the triangle or in the middle of the triangle.

probHMM.py — (continued)

```
# state
#       0=middle, 1,2,3 are corners
states1 = {'middle', 'c1', 'c2', 'c3'} # states
obs1 = {'m1','m2','m3'} # microphones
```

The observation model is as follows. If the animal is in a corner, it will be detected by the microphone at that corner with probability 0.6, and will be independently detected by each of the other microphones with a probability of 0.1. If the animal is in the middle, it will be detected by each microphone with a probability of 0.4.

probHMM.py — (continued)

```
# pobs gives the observation model:
#pobs[mi][state] is P(mi=on | state)
closeMic=0.6; farMic=0.1; midMic=0.4
pobs1 = {'m1':{'middle':midMic, 'c1':closeMic, 'c2':farMic, 'c3':farMic},
    # mic 1
        'm2':{'middle':midMic, 'c1':farMic, 'c2':closeMic, 'c3':farMic}, #
            mic 2
        'm3':{'middle':midMic, 'c1':farMic, 'c2':farMic, 'c3':closeMic}} #
            mic 3
```

The transition model is as follows: If the animal is in a corner it stays in the same corner with probability 0.80, goes to the middle with probability 0.1 or goes to one of the other corners with probability 0.05 each. If it is in the middle, it stays in the middle with probability 0.7, otherwise it moves to one the corners, each with probability 0.1.

probHMM.py — (continued)

```
# trans specifies the dynamics
# trans[i] is the distribution over states resulting from state i
# trans[i][j] gives P(S=j | S=i)
sm=0.7; mmc=0.1             # transition probabilities when in middle
sc=0.8; mcm=0.1; mcc=0.05   # transition probabilities when in a corner
trans1 = {'middle':{'middle':sm, 'c1':mmc, 'c2':mmc, 'c3':mmc}, # was in
    middle
         'c1':{'middle':mcm, 'c1':sc, 'c2':mcc, 'c3':mcc}, # was in corner
             1
         'c2':{'middle':mcm, 'c1':mcc, 'c2':sc, 'c3':mcc}, # was in corner
             2
         'c3':{'middle':mcm, 'c1':mcc, 'c2':mcc, 'c3':sc}} # was in corner
             3
```

Initially the animal is in one of the four states, with equal probability.

probHMM.py — (continued)

```
# initially we have a uniform distribution over the animal's state
indist1 = {st:1.0/len(states1) for st in states1}

hmm1 = HMM(states1, obs1, pobs1, trans1, indist1)
```

9.10.1 Exact Filtering for HMMs

A *HMMVEfilter* has a current state distribution which can be updated by observing or by advancing to the next time.

probHMM.py — (continued)

```
from display import Displayable

class HMMVEfilter(Displayable):
    def __init__(self,hmm):
        self.hmm = hmm
        self.state_dist = hmm.indist

    def filter(self, obsseq):
        """updates and returns the state distribution following the
            sequence of
        observations in obsseq using variable elimination.

        Note that it first advances time.
        This is what is required if it is called sequentially.
        If that is not what is wanted initially, do an observe first.
        """
        for obs in obsseq:
            self.advance()    # advance time
            self.observe(obs) # observe
        return self.state_dist

    def observe(self, obs):
        """updates state conditioned on observations.
        obs is a list of values for each observation variable"""
        for i in self.hmm.obsvars:
            self.state_dist = {st:self.state_dist[st]*(self.hmm.pobs[i][st]
                                                if obs[i] else
                                                    (1-self.hmm.pobs[i][st]))
                               for st in self.hmm.states}
        norm = sum(self.state_dist.values()) # normalizing constant
        self.state_dist = {st:self.state_dist[st]/norm for st in
            self.hmm.states}
        self.display(2,"After observing",obs,"state
            distribution:",self.state_dist)

    def advance(self):
        """advance to the next time"""
        nextstate = {st:0.0 for st in self.hmm.states}  # distribution over
            next states
```

```
90          for j in self.hmm.states:     # j ranges over next states
91              for i in self.hmm.states: # i ranges over previous states
92                  nextstate[j] += self.hmm.trans[i][j]*self.state_dist[i]
93          self.state_dist = nextstate
94          self.display(2,"After advancing state
                distribution:",self.state_dist)
```

The following are some queries for *hmm*1.

probHMM.py — (continued)

```
96  hmm1f1 = HMMVEfilter(hmm1)
97  # hmm1f1.filter([{'m1':0, 'm2':1, 'm3':1}, {'m1':1, 'm2':0, 'm3':1}])
98  ## HMMVEfilter.max_display_level = 2 # show more detail in displaying
99  # hmm1f2 = HMMVEfilter(hmm1)
100 # hmm1f2.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':1, 'm3':0},
        {'m1':1, 'm2':0, 'm3':0},
101 #              {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
        {'m1':0, 'm2':0, 'm3':0},
102 #              {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':1},
        {'m1':0, 'm2':0, 'm3':1},
103 #              {'m1':0, 'm2':0, 'm3':1}])
104 # hmm1f3 = HMMVEfilter(hmm1)
105 # hmm1f3.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
        {'m1':1, 'm2':0, 'm3':0}, {'m1':1, 'm2':0, 'm3':1}])
106
107 # How do the following differ in the resulting state distribution?
108 # Note they start the same, but have different initial observations.
109 ## HMMVEfilter.max_display_level = 1 # show less detail in displaying
110 # for i in range(100): hmm1f1.advance()
111 # hmm1f1.state_dist
112 # for i in range(100): hmm1f3.advance()
113 # hmm1f3.state_dist
```

Exercise 9.7 The representation assumes that there are a list of Boolean observations. Extend the representation so that the each observation variable can have multiple discrete values. You need to choose a representation for the model, and change the algorithm.

9.10.2 Localization

The localization example in the book is a controlled HMM, where there is a given action at each time and the transition depends on the action.

probLocalization.py — Controlled HMM and Localization example

```
11  from probHMM import HMMVEfilter, HMM
12  from display import Displayable
13  import matplotlib.pyplot as plt
14  from matplotlib.widgets import Button, CheckButtons
15
16  class HMM_Controlled(HMM):
```

```
    """A controlled HMM, where the transition probability depends on the
        action.
       Instead of the transition probability, it has a function act2trans
       from action to transition probability.
       Any algorithms need to select the transition probability according
           to the action.
    """
    def __init__(self, states, obsvars, pobs, act2trans, indist):
        self.act2trans = act2trans
        HMM.__init__(self, states, obsvars, pobs, None, indist)


local_states = list(range(16))
door_positions = {2,4,7,11}
def prob_door(loc): return 0.8 if loc in door_positions else 0.1
local_obs = {'door':[prob_door(i) for i in range(16)]}
act2trans = {'right': [[0.1 if next == current
                       else 0.8 if next == (current+1)%16
                       else 0.074 if next == (current+2)%16
                       else 0.002 for next in range(16)]
                          for current in range(16)],
             'left': [[0.1 if next == current
                       else 0.8 if next == (current-1)%16
                       else 0.074 if next == (current-2)%16
                       else 0.002 for next in range(16)]
                         for current in range(16)]}
hmm_16pos = HMM_Controlled(local_states, {'door'}, local_obs,
                           act2trans, [1/16 for i in range(16)])
```

To change the VE localization code to allow for controlled HMMs, notice that the action selects which transition probability to us.

probLocalization.py — (continued)

```
class HMM_Local(HMMVEfilter):
    """VE filter for controlled HMMs
    """
    def __init__(self, hmm):
        HMMVEfilter.__init__(self, hmm)

    def go(self, action):
        self.hmm.trans = self.hmm.act2trans[action]
        self.advance()

loc_filt = HMM_Local(hmm_16pos)
# loc_filt.observe({'door':True}); loc_filt.go("right");
    loc_filt.observe({'door':False}); loc_filt.go("right");
    loc_filt.observe({'door':True})
# loc_filt.state_dist
```

The following lets us interactively move the agent and provide observations. It shows the distribution over locations. Figure 9.8 shows the GUI ob-

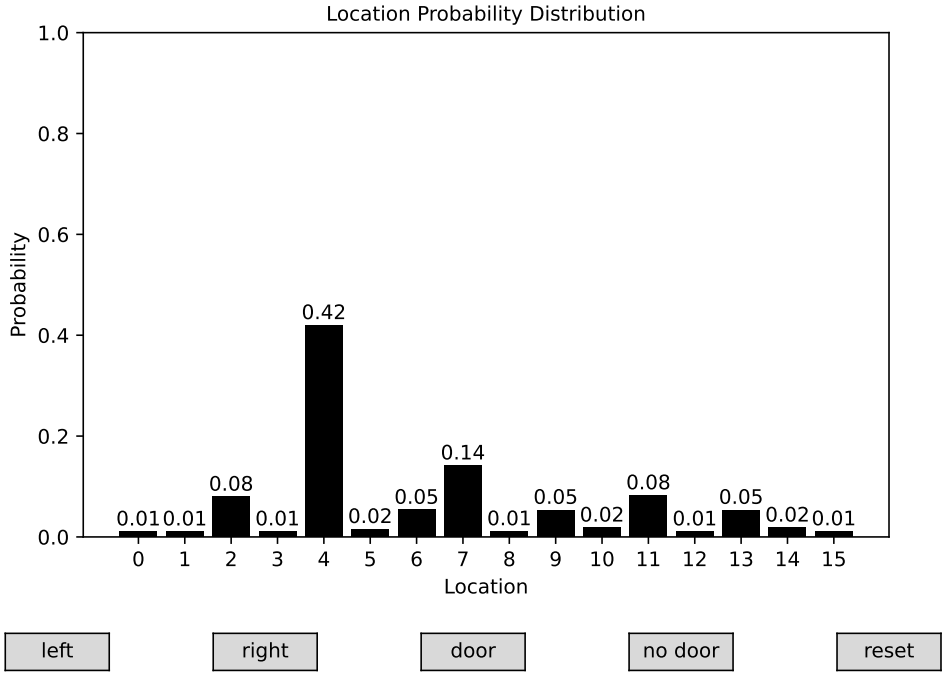


Figure 9.8: Localization GUI after observing a door, moving right, observing no door, moving right, and observing a door.

tained by `Show_Localization(hmm_16pos)` after some interaction.

probLocalization.py — (continued)

```
57 class Show_Localization(Displayable):
58     def __init__(self, hmm, fontsize=10):
59         self.hmm = hmm
60         self.fontsize = fontsize
61         self.loc_filt = HMM_Local(hmm)
62         fig, self.ax = plt.subplots()
63         fig.subplots_adjust(bottom=0.2)
64         ## Set up buttons:
65         left_butt = Button(fig.add_axes([0.05,0.02,0.1,0.05]), "left")
66         left_butt.label.set_fontsize(self.fontsize)
67         left_butt.on_clicked(self.left)
68         right_butt = Button(fig.add_axes([0.25,0.02,0.1,0.05]), "right")
69         right_butt.label.set_fontsize(self.fontsize)
70         right_butt.on_clicked(self.right)
71         door_butt = Button(fig.add_axes([0.45,0.02,0.1,0.05]), "door")
72         door_butt.label.set_fontsize(self.fontsize)
73         door_butt.on_clicked(self.door)
74         nodoor_butt = Button(fig.add_axes([0.65,0.02,0.1,0.05]), "no door")
75         nodoor_butt.label.set_fontsize(self.fontsize)
```

```
        nodoor_butt.on_clicked(self.nodoor)
        reset_butt = Button(fig.add_axes([0.85,0.02,0.1,0.05]), "reset")
        reset_butt.label.set_fontsize(self.fontsize)
        reset_butt.on_clicked(self.reset)
        ## draw the distribution
        plt.subplot(1, 1, 1)
        self.draw_dist()
        plt.show()

    def draw_dist(self):
        self.ax.clear()
        self.ax.set_ylim(0,1)
        self.ax.set_ylabel("Probability", fontsize=self.fontsize)
        self.ax.set_xlabel("Location", fontsize=self.fontsize)
        self.ax.set_title("Location Probability Distribution",
            fontsize=self.fontsize)
        self.ax.set_xticks(self.hmm.states, labels = self.hmm.states,
            fontsize=self.fontsize)
        vals = [self.loc_filt.state_dist[i] for i in self.hmm.states]
        self.bars = self.ax.bar(self.hmm.states, vals, color='black')
        self.ax.bar_label(self.bars,["{v:.2f}".format(v=v) for v in vals],
            padding = 1, fontsize=self.fontsize)
        plt.draw()

    def left(self,event):
        self.loc_filt.go("left")
        self.draw_dist()
    def right(self,event):
        self.loc_filt.go("right")
        self.draw_dist()
    def door(self,event):
        self.loc_filt.observe({'door':True})
        self.draw_dist()
    def nodoor(self,event):
        self.loc_filt.observe({'door':False})
        self.draw_dist()
    def reset(self,event):
        self.loc_filt.state_dist = {i:1/16 for i in range(16)}
        self.draw_dist()

# Show_Localization(hmm_16pos)
# Show_Localization(hmm_16pos, fontsize=15) # for demos - enlarge window

if __name__ == "__main__":
    print("Try: Show_Localization(hmm_16pos)")
```

9.10.3 Particle Filtering for HMMs

In this implementation, a particle is just a state. If you want to do some form of smoothing, a particle should probably be a history of states. This maintains, *particles*, an array of states, *weights* an array of (non-negative) real numbers, such that $weights[i]$ is the weight of $particles[i]$.

probHMM.py — (continued)

```
114 from display import Displayable
115 from probStochSim import resample
116
117 class HMMparticleFilter(Displayable):
118     def __init__(self,hmm,number_particles=1000):
119         self.hmm = hmm
120         self.particles = [sample_one(hmm.indist)
121                           for i in range(number_particles)]
122         self.weights = [1 for i in range(number_particles)]
123
124     def filter(self, obsseq):
125         """returns the state distribution following the sequence of
126         observations in obsseq using particle filtering.
127
128         Note that it first advances time.
129         This is what is required if it is called after previous filtering.
130         If that is not what is wanted initially, do an observe first.
131         """
132         for obs in obsseq:
133             self.advance()   # advance time
134             self.observe(obs) # observe
135             self.resample_particles()
136             self.display(2,"After observing", str(obs),
137                          "state distribution:",
                                 self.histogram(self.particles))
138         self.display(1,"Final state distribution:",
                self.histogram(self.particles))
139         return self.histogram(self.particles)
140
141     def advance(self):
142         """advance to the next time.
143         This assumes that all of the weights are 1."""
144         self.particles = [sample_one(self.hmm.trans[st])
145                           for st in self.particles]
146
147     def observe(self, obs):
148         """reweighs the particles to incorporate observations obs"""
149         for i in range(len(self.particles)):
150             for obv in obs:
151                 if obs[obv]:
152                     self.weights[i] *= self.hmm.pobs[obv][self.particles[i]]
153                 else:
```

```
                self.weights[i] *=
                    1-self.hmm.pobs[obv][self.particles[i]]

    def histogram(self, particles):
        """returns list of the probability of each state as represented by
        the particles"""
        tot=0
        hist = {st: 0.0 for st in self.hmm.states}
        for (st,wt) in zip(self.particles,self.weights):
            hist[st]+=wt
            tot += wt
        return {st:hist[st]/tot for st in hist}

    def resample_particles(self):
        """resamples to give a new set of particles."""
        self.particles = resample(self.particles, self.weights,
            len(self.particles))
        self.weights = [1] * len(self.particles)
```

The following are some queries for *hmm*1.

probHMM.py — (continued)

```
hmm1pf1 = HMMparticleFilter(hmm1)
# HMMparticleFilter.max_display_level = 2 # show each step
# hmm1pf1.filter([{'m1':0, 'm2':1, 'm3':1}, {'m1':1, 'm2':0, 'm3':1}])
# hmm1pf2 = HMMparticleFilter(hmm1)
# hmm1pf2.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':1, 'm3':0},
    {'m1':1, 'm2':0, 'm3':0},
#               {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
    {'m1':0, 'm2':0, 'm3':0},
#               {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':1},
    {'m1':0, 'm2':0, 'm3':1},
#               {'m1':0, 'm2':0, 'm3':1}])
# hmm1pf3 = HMMparticleFilter(hmm1)
# hmm1pf3.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
    {'m1':1, 'm2':0, 'm3':0}, {'m1':1, 'm2':0, 'm3':1}])
```

Exercise 9.8 A form of importance sampling can be obtained by not resampling. Is it better or worse than particle filtering? Hint: you need to think about how they can be compared. Is the comparison different if there are more states than particles?

Exercise 9.9 Extend the particle filtering code to continuous variables and observations. In particular, suppose the state transition is a linear function with Gaussian noise of the previous state, and the observations are linear functions with Gaussian noise of the state. You may need to research how to sample from a Gaussian distribution (or use Python's `random` library) .

9.10.4 Generating Examples

The following code is useful for generating examples.

probHMM.py — (continued)

```
182 def simulate(hmm,horizon):
183     """returns a pair of (state sequence, observation sequence) of length
            horizon.
184     for each time t, the agent is in state_sequence[t] and
185     observes observation_sequence[t]
186     """
187     state = sample_one(hmm.indist)
188     obsseq=[]
189     stateseq=[]
190     for time in range(horizon):
191         stateseq.append(state)
192         newobs =
                {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
193                  for obs in hmm.obsvars}
194         obsseq.append(newobs)
195         state = sample_one(hmm.trans[state])
196     return stateseq,obsseq
197
198 def simobs(hmm,stateseq):
199     """returns observation sequence for the state sequence"""
200     obsseq=[]
201     for state in stateseq:
202         newobs =
                {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
203                  for obs in hmm.obsvars}
204         obsseq.append(newobs)
205     return obsseq
206
207 def create_eg(hmm,n):
208     """Create an annotated example for horizon n"""
209     seq,obs = simulate(hmm,n)
210     print("True state sequence:",seq)
211     print("Sequence of observations:\n",obs)
212     hmmfilter = HMMVEfilter(hmm)
213     dist = hmmfilter.filter(obs)
214     print("Resulting distribution over states:\n",dist)
```

9.11 Dynamic Belief Networks

A **dynamic belief network (DBN)** is a belief network that extends in time.

There are a number of ways that reasoning can be carried out in a DBN, including:

- Rolling out the DBN for some time period, and using standard belief network inference. The latest time that needs to be in the rolled out network is the time of the latest observation or the time of a query (whichever is

later). This allows us to observe any variables at any time and query any variables at any time. This is covered in Section 9.11.2.

- An unrolled belief network may be very large, and we might only be interested in asking about "now". In this case we can just representing the variables "now". In this approach we can observe and query the current variables. We can them move to the next time. This does not allow for arbitrary historical queries (about the past or the future), but can be much simpler. This is covered in Section 9.11.3.

9.11.1 Representing Dynamic Belief Networks

To specify a DBN, consider an arbitrary point, *now*, which will will be represented as time 1. Each variable will have a corresponding previous variable; the variables and their previous instances will be created together.

A dynamic belief network consists of:

- A set of features. A variable is a feature-time pair.
- An initial distribution over the features "now" (time 1). This is a belief network with all variables being time 1 variables.
- A specification of the dynamics. We define the how the variables *now* (time 1) depend on variables *now* and the previous time (time 0), in such a way that the graph is acyclic.

probDBN.py — Dynamic belief networks

```
from variable import Variable
from probGraphicalModels import GraphicalModel, BeliefNetwork
from probFactors import Prob, Factor, CPD
from probVE import VE
from display import Displayable

class DBNvariable(Variable):
    """A random variable that incorporates the stage (time)

    A DBN variable has both a name and an index. The index defaults to 1.
    position is (x,y) where x>0.3
    """
    def __init__(self, name, domain=[False,True], index=1, position=None):
        Variable.__init__(self, f"{name}_{index}", domain,
            position=position)
        self.basename = name
        self.domain = domain
        self.index = index
        self.previous = None

    def __lt__(self,other):
```

```
31          if self.name == other.name:
32              return self.index < other.index
33          else:
34              return self.name < other.name
35
36  def variable_pair(name, domain=[False,True], position=None):
37      """returns a variable and its predecessor. This is used to define
            2-stage DBNs
38
39      If the name is X, it returns the pair of variables X_prev,X_now"""
40      var_now = DBNvariable(name, domain, index='now', position=position)
41      if position:
42          (x,y) = position
43          position = (x-0.3, y)
44      var_prev = DBNvariable(name, domain, index='prev', position=position)
45      var_now.previous = var_prev
46      return var_prev, var_now
```

A *FactorRename* is a factor that is the result of renaming the variables in the factor. It takes a factor, *fac,* and a $\{new : old\}$ dictionary, where *new* is the name of a variable in the resulting factor and *old* is the corresponding name in *fac*. This assumes that all variables are renamed.

probDBN.py — (continued)

```
48  class FactorRename(Factor):
49      def __init__(self,fac,renaming):
50          """A renamed factor.
51          fac is a factor
52          renaming is a dictionary of the form {new:old} where old and new
                var variables,
53             where the variables in fac appear exactly once in the renaming
54          """
55          Factor.__init__(self,[n for (n,o) in renaming.items() if o in
                fac.variables])
56          self.orig_fac = fac
57          self.renaming = renaming
58
59      def get_value(self,assignment):
60          return self.orig_fac.get_value({self.renaming[var]:val
61                                          for (var,val) in assignment.items()
62                                          if var in self.variables})
```

The following class renames the variables of a conditional probability distribution. It is used for template models (e.g., dynamic decision networks or relational models)

probDBN.py — (continued)

```
64  class CPDrename(FactorRename, CPD):
65      def __init__(self, cpd, renaming):
66          renaming_inverse = {old:new for (new,old) in renaming.items()}
```

```
        CPD.__init__(self,renaming_inverse[cpd.child],[renaming_inverse[p]
            for p in cpd.parents])
        self.orig_fac = cpd
        self.renaming = renaming
```

probDBN.py — (continued)

```
class DBN(Displayable):
    """The class of stationary Dynamic Belief networks.
    * name is the DBN name
    * vars_now is a list of current variables (each must have
    previous variable).
    * transition_factors is a list of factors for P(X|parents) where X
    is a current variable and parents is a list of current or previous
        variables.
    * init_factors is a list of factors for P(X|parents) where X is a
    current variable and parents can only include current variables
    The graph of transition factors + init factors must be acyclic.

    """
    def __init__(self, title, vars_now, transition_factors=None,
        init_factors=None):
        self.title = title
        self.vars_now = vars_now
        self.vars_prev = [v.previous for v in vars_now]
        self.transition_factors = transition_factors
        self.init_factors = init_factors
        self.var_index = {}    # var_index[v] is the index of variable v
        for i,v in enumerate(vars_now):
            self.var_index[v]=i

    def show(self):
        BNfromDBN(self,1).show()
```

Here is a 3 variable DBN (shown in Figure 9.9):

probDBN.py — (continued)

```
A0,A1 = variable_pair("A", domain=[False,True], position = (0.4,0.8))
B0,B1 = variable_pair("B", domain=[False,True], position = (0.4,0.5))
C0,C1 = variable_pair("C", domain=[False,True], position = (0.4,0.2))

# dynamics
pc = Prob(C1,[B1,C0],[[[0.03,0.97],[0.38,0.62]],[[0.23,0.77],[0.78,0.22]]])
pb = Prob(B1,[A0,A1],[[[0.5,0.5],[0.77,0.23]],[[0.4,0.6],[0.83,0.17]]])
pa = Prob(A1,[A0,B0],[[[0.1,0.9],[0.65,0.35]],[[0.3,0.7],[0.8,0.2]]])

# initial distribution
pa0 = Prob(A1,[],[0.9,0.1])
pb0 = Prob(B1,[A1],[[0.3,0.7],[0.8,0.2]])
pc0 = Prob(C1,[],[0.2,0.8])

dbn1 = DBN("Simple DBN",[A1,B1,C1],[pa,pb,pc],[pa0,pb0,pc0])
```

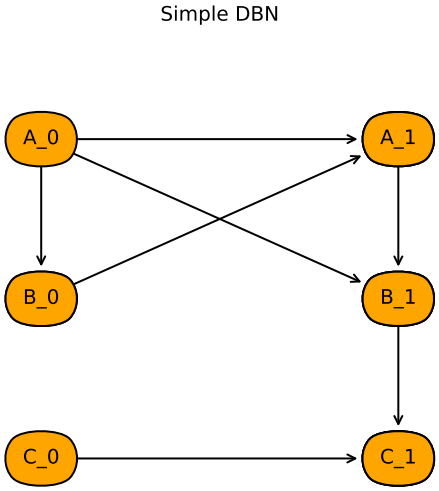


Figure 9.9: Simple dynamic belief network (dbn1.show())

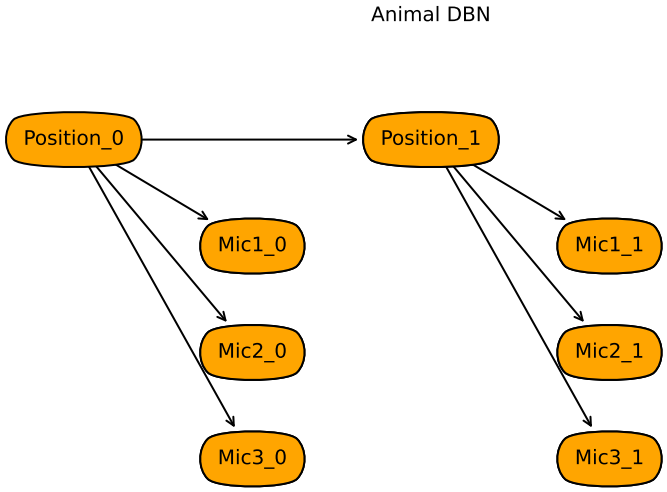


Figure 9.10: Animal dynamic belief network (dbn_an.show())

Here is the animal example

probDBN.py — (continued)

```
from probHMM import closeMic, farMic, midMic, sm, mmc, sc, mcm, mcc

Pos_0,Pos_1 = variable_pair("Position", domain=[0,1,2,3],
    position=(0.5,0.8))
Mic1_0,Mic1_1 = variable_pair("Mic1", position=(0.6,0.6))
Mic2_0,Mic2_1 = variable_pair("Mic2", position=(0.6,0.4))
Mic3_0,Mic3_1 = variable_pair("Mic3", position=(0.6,0.2))

# conditional probabilities - see hmm for the values of sm,mmc, etc
ppos = Prob(Pos_1, [Pos_0],
           [[sm, mmc, mmc, mmc], #was in middle
            [mcm, sc, mcc, mcc], #was in corner 1
            [mcm, mcc, sc, mcc], #was in corner 2
            [mcm, mcc, mcc, sc]]) #was in corner 3
pm1 = Prob(Mic1_1, [Pos_1], [[1-midMic, midMic], [1-closeMic, closeMic],
                          [1-farMic, farMic], [1-farMic, farMic]])
pm2 = Prob(Mic2_1, [Pos_1], [[1-midMic, midMic], [1-farMic, farMic],
                          [1-closeMic, closeMic], [1-farMic, farMic]])
pm3 = Prob(Mic3_1, [Pos_1], [[1-midMic, midMic], [1-farMic, farMic],
                          [1-farMic, farMic], [1-closeMic, closeMic]])
ipos = Prob(Pos_1,[], [0.25, 0.25, 0.25, 0.25])
dbn_an =DBN("Animal DBN",[Pos_1,Mic1_1,Mic2_1,Mic3_1],
           [ppos, pm1, pm2, pm3],
           [ipos, pm1, pm2, pm3])
```

9.11.2 Unrolling DBNs

probDBN.py — (continued)

```
class BNfromDBN(BeliefNetwork):
    """Belief Network unrolled from a dynamic belief network
    """

    def __init__(self,dbn,horizon):
        """dbn is the dynamic belief network being unrolled
        horizon>0 is the number of steps (so there will be horizon+1
            variables for each DBN variable.
        """
        self.dbn = dbn
        self.horizon = horizon
        self.minx,self.width = None, None # for positions pf variables
        self.name2var = {var.basename:
            [DBNvariable(var.basename,var.domain,index,
                                                  position=self.pos(var,index))
                                          for index in range(horizon+1)]
                        for var in dbn.vars_now}
        self.display(1,f"name2var={self.name2var}")
```

```
        variables = {v for vs in self.name2var.values() for v in vs}
        self.display(1,f"variables={variables}")
        bnfactors = {CPDrename(fac,{self.name2var[var.basename][0]:var
                                    for var in fac.variables})
                     for fac in dbn.init_factors}
        bnfactors |= {CPDrename(fac,{self.name2var[var.basename][i]:var
                                    for var in fac.variables if
                                        var.index=='prev'}
                                  | {self.name2var[var.basename][i+1]:var
                                    for var in fac.variables if
                                        var.index=='now'})
                     for fac in dbn.transition_factors
                        for i in range(horizon)}
        self.display(1,f"bnfactors={bnfactors}")
        BeliefNetwork.__init__(self, dbn.title, variables, bnfactors)

    def pos(self, var, index):
        minx = min(x for (x,y) in (var.position for var in
            self.dbn.vars_now))-1e-6
        maxx = max(x for (x,y) in (var.position for var in
            self.dbn.vars_now))
        width = maxx-minx
        xo,yo = var.position
        xi = index/(self.horizon+1)+(xo-minx)/width/(self.horizon+1)/2
        return (xi, yo)
```

Here are two examples. You use `bn.name2var['B'][2]` to get the variable B2 (B at time 2). Figure 9.11 shows the output of the `drc.show_post` below:

probDBN.py — (continued)

```
# Try
from probRC import ProbRC
# bn = BNfromDBN(dbn1,2) # construct belief network
# drc = ProbRC(bn)            # initialize recursive conditioning
# B2 = bn.name2var['B'][2]
# drc.query(B2) #P(B2)
#
    drc.query(bn.name2var['B'][1],{bn.name2var['B'][0]:True,bn.name2var['C'][1]:False})
    #P(B1|b0,~c1)
# drc.show_post({bn.name2var['B'][0]:True,bn.name2var['C'][1]:False})

# Plot Distributions:
# bna = BNfromDBN(dbn_an,5) # animal belief network with horizon 5
# dra = ProbRC(bna)
# dra.show_post(obs =
    {bna.name2var['Mic1'][1]:True,bna.name2var['Mic1'][2]:True})
```

9.11.3 DBN Filtering

If we only wanted to ask questions about the current state, we can save space by forgetting the history variables.

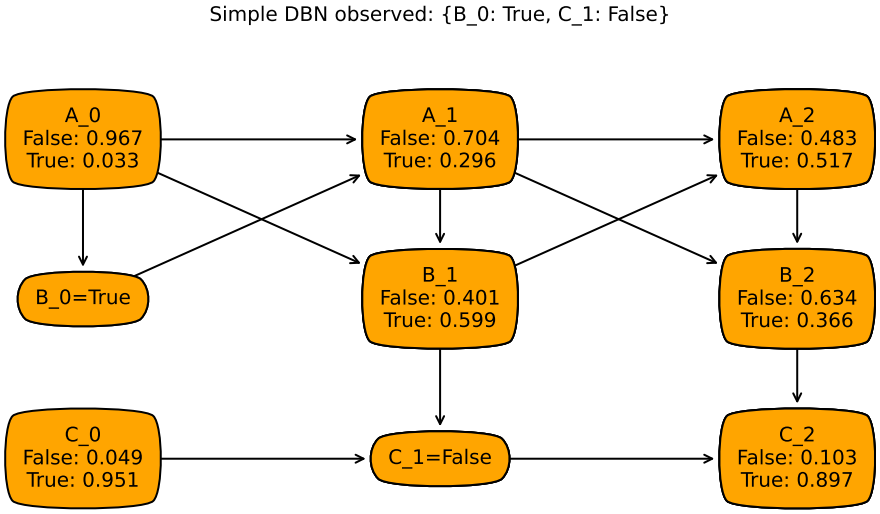


Figure 9.11: Simple dynamic belief network (dbn1) horizon 2

probDBN.py — (continued)

```
188 class DBNVEfilter(VE):
189     def __init__(self,dbn):
190         self.dbn = dbn
191         self.current_factors = dbn.init_factors
192         self.current_obs = {}
193 
194     def observe(self, obs):
195         """updates the current observations with obs.
196         obs is a variable:value dictionary where variable is a current
197         variable.
198         """
199         assert all(self.current_obs[var]==obs[var] for var in obs
200                    if var in self.current_obs),"inconsistent current
                           observations"
201         self.current_obs.update(obs) # note 'update' is a dict method
202 
203     def query(self,var):
204         """returns the posterior probability of current variable var"""
205         return
                VE(GraphicalModel(self.dbn.title,self.dbn.vars_now,self.current_factors)
206                    ).query(var,self.current_obs)
207 
208     def advance(self):
```

```
        """advance to the next time"""
        prev_factors = [self.make_previous(fac) for fac in
            self.current_factors]
        prev_obs = {var.previous:val for var,val in
            self.current_obs.items()}
        two_stage_factors = prev_factors + self.dbn.transition_factors
        self.current_factors =
            self.elim_vars(two_stage_factors,self.dbn.vars_prev,prev_obs)
        self.current_obs = {}

    def make_previous(self,fac):
         """Creates new factor from fac where the current variables in fac
         are renamed to previous variables.
         """
         return FactorRename(fac, {var.previous:var for var in
             fac.variables})

    def elim_vars(self,factors, vars, obs):
        for var in vars:
            if var in obs:
                factors = [self.project_observations(fac,obs) for fac in
                    factors]
            else:
                factors = self.eliminate_var(factors, var)
        return factors
```

Example queries:

probDBN.py — (continued)

```
#df = DBNVEfilter(dbn1)
#df.observe({B1:True}); df.advance(); df.observe({C1:False})
#df.query(B1) #P(B1|B0,C1)
#df.advance(); df.query(B1)
#dfa = DBNVEfilter(dbn_an)
# dfa.observe({Mic1_1:0, Mic2_1:1, Mic3_1:1})
# dfa.advance()
# dfa.observe({Mic1_1:1, Mic2_1:0, Mic3_1:1})
# dfa.query(Pos_1)
```

Chapter 10

Learning with Uncertainty

10.1 Bayesian Learning

The section contains two implementations of the (discretized) beta distribution. The first represents Bayesian learning as a belief network. The second is an interactive tool to understand the beta distribution.

The following uses a belief network representation from the previous chapter to learn (discretized) probabilities. Figure 10.1 shows the output after observing *heads, heads, tails*. Notice the prediction of future tosses.

learnBayesian.py — Bayesian Learning

```
11 from variable import Variable
12 from probFactors import Prob
13 from probGraphicalModels import BeliefNetwork
14 from probRC import ProbRC
15
16 #### Coin Toss ###
17 # multiple coin tosses:
18 toss = ['tails','heads']
19 tosses = [ Variable(f"Toss#{i}", toss,
20                     (0.8, 0.9-i/10) if i<10 else (0.4,0.2))
21                for i in range(11)]
22
23 def coinTossBN(num_bins = 10):
24     prob_bins = [x/num_bins for x in range(num_bins+1)]
25     PH = Variable("P_heads", prob_bins, (0.1,0.9))
26     p_PH = Prob(PH,[],{x:0.5/num_bins if x in [0,1] else 1/num_bins for x
           in prob_bins})
27     p_tosses = [ Prob(tosses[i],[PH], {x:{'tails':1-x,'heads':x} for x in
           prob_bins})
28                for i in range(11)]
```

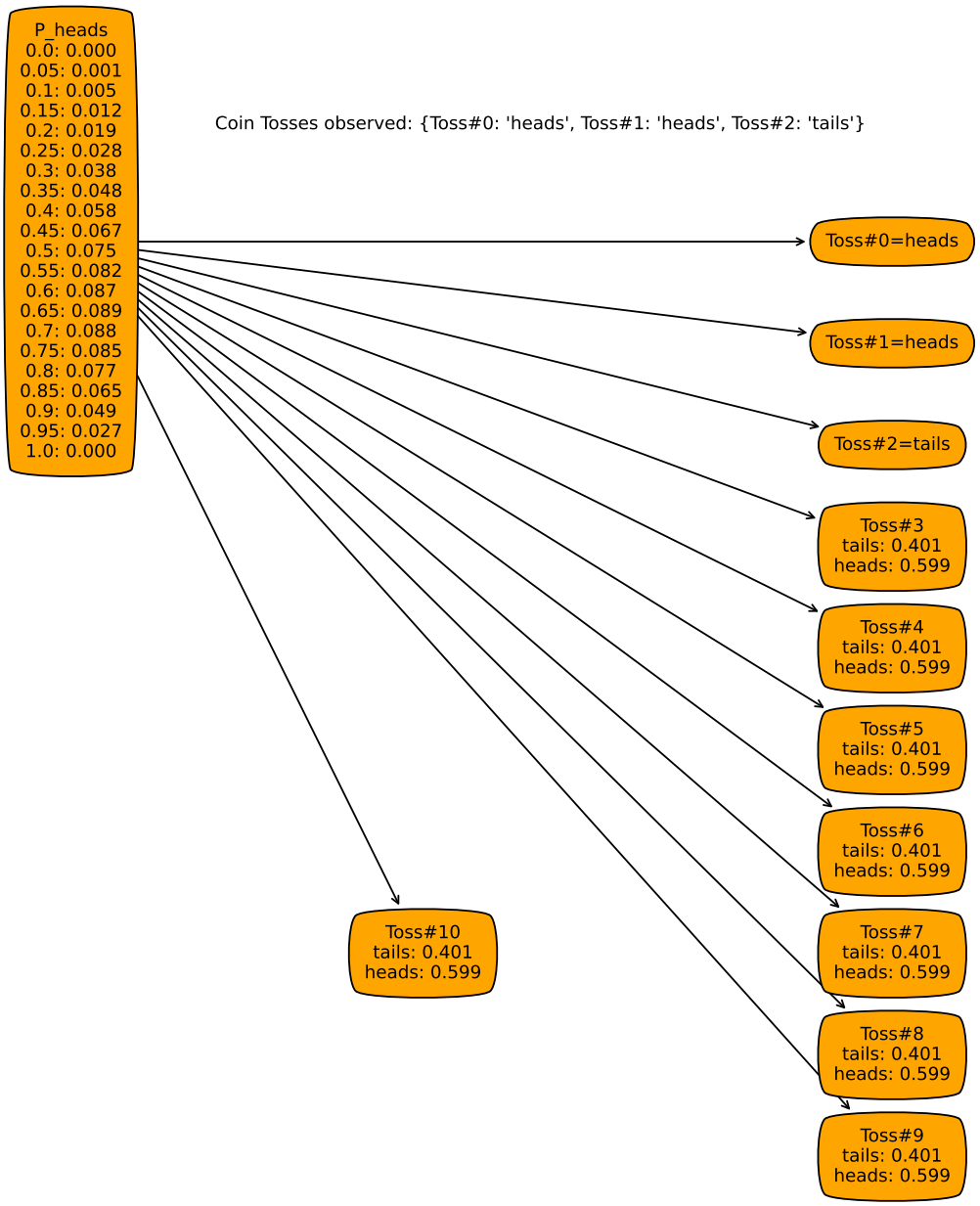


Figure 10.1: coinTossBN after observing heads, heads, tails

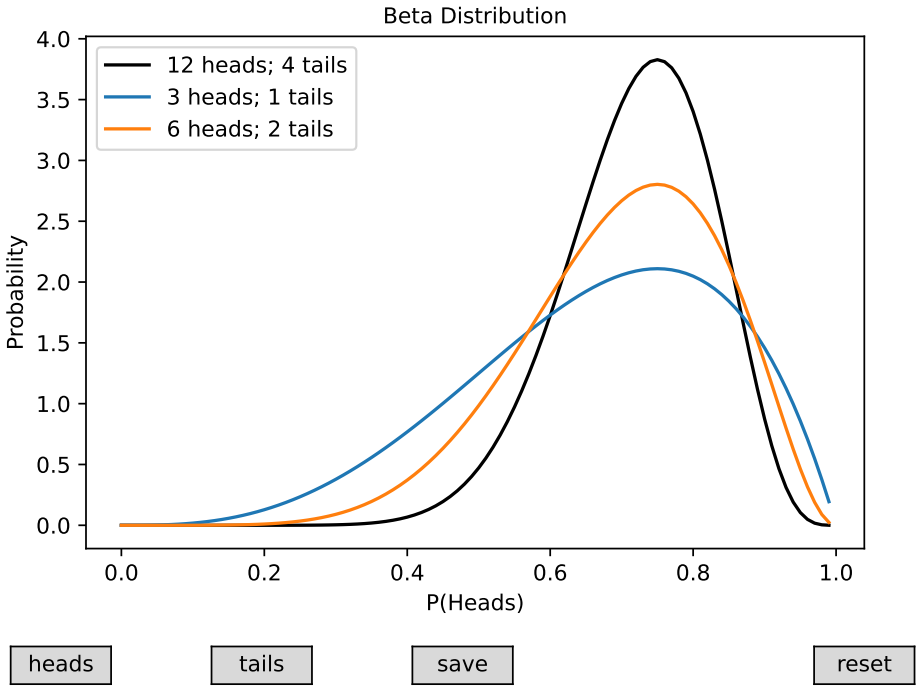


Figure 10.2: Beta distribution after some observations

```
    return BelifNetwork("Coin Tosses",
                        [PH]+tosses,
                        [p_PH]+p_tosses)


#
# coinRC = ProbRC(coinTossBN(20))
# coinRC.query(tosses[10],{tosses[0]:'heads'})
# coinRC.show_post({})
# coinRC.show_post({tosses[0]:'heads'})
# coinRC.show_post({tosses[0]:'heads',tosses[1]:'heads'})
# coinRC.show_post({tosses[0]:'heads',tosses[1]:'heads',tosses[2]:'tails'})
```

Figure 10.2 shows a plot of the Beta distribution (the *P_head* variable in the previous belief network) given some sets of observations.

This is a plot that is produced by the following interactive tool.

learnBayesian.py — (continued)

```
from display import Displayable
import matplotlib.pyplot as plt
from matplotlib.widgets import Button, CheckButtons

class Show_Beta(Displayable):
```

```
    def __init__(self,num=100, fontsize=10):
        self.num = num
        self.dist = [1 for i in range(num)]
        self.vals = [i/num for i in range(num)]
        self.fontsize = fontsize
        self.saves = []
        self.num_heads = 0
        self.num_tails = 0
        plt.ioff()
        fig, self.ax = plt.subplots()
        plt.subplots_adjust(bottom=0.2)
        ## Set up buttons:
        heads_butt = Button(fig.add_axes([0.05,0.02,0.1,0.05]), "heads")
        heads_butt.label.set_fontsize(self.fontsize)
        heads_butt.on_clicked(self.heads)
        tails_butt = Button(fig.add_axes([0.25,0.02,0.1,0.05]), "tails")
        tails_butt.label.set_fontsize(self.fontsize)
        tails_butt.on_clicked(self.tails)
        save_butt = Button(fig.add_axes ([0.45,0.02,0.1,0.05]), "save")
        save_butt.label.set_fontsize(self.fontsize)
        save_butt.on_clicked(self.save)
        reset_butt = Button(fig.add_axes ([0.85,0.02,0.1,0.05]), "reset")
        reset_butt.label.set_fontsize(self.fontsize)
        reset_butt.on_clicked(self.reset)
        ## draw the distribution
        self.draw_dist()
        plt.show()

    def draw_dist(self):
        sv = self.num/sum(self.dist)
        self.dist = [v*sv for v in self.dist]
        #print(self.dist)
        self.ax.clear()
        self.ax.set_ylabel("Probability", fontsize=self.fontsize)
        self.ax.set_xlabel("P(Heads)", fontsize=self.fontsize)
        self.ax.set_title("Beta Distribution", fontsize=self.fontsize)
        self.ax.plot(self.vals, self.dist, color='black', label =
            f"{self.num_heads} heads; {self.num_tails} tails")
        for (nh,nt,d) in self.saves:
            self.ax.plot(self.vals, d, label = f"{nh} heads; {nt} tails")
        self.ax.legend()
        plt.draw()

    def heads(self,event):
        self.num_heads += 1
        self.dist = [self.dist[i]*self.vals[i] for i in range(self.num)]
        self.draw_dist()
    def tails(self,event):
        self.num_tails += 1
        self.dist = [self.dist[i]*(1-self.vals[i]) for i in range(self.num)]
```

```
96          self.draw_dist()
97      def save(self,event):
98          self.saves.append((self.num_heads,self.num_tails,self.dist))
99          self.draw_dist()
100     def reset(self,event):
101         self.num_tails = 0
102         self.num_heads = 0
103         self.dist = [1/self.num for i in range(self.num)]
104         self.draw_dist()
105
106 # s1 = Show_Beta(100)
107 # sl = Show_Beta(100, fontsize=15) # for demos - enlarge window
108
109 if __name__ == "__main__":
110     print("Try: Show_Beta(100)")
```

10.2 K-means

The k-means learner takes in a dataset and a number of classes, and learns a mapping from examples to classes (`class_of_eg`) and a function that makes predictions for classes (`class_predictions`).

It maintains two lists that suffice as sufficient statistics to classify examples, and to learn the classification:

- *class_counts* is a list such that $class\_counts[c]$ is the number of examples in the training set with $class = c$.
- *feature_sum* is a list such that $feature\_sum[f][c]$ is sum of the values for the feature f for members of class c. The average value of the ith feature in class i is

$$\frac{feature\_sum[i][c]}{class\_counts[c]}$$

  when $class\_counts[c] > 0$ and is 0 otherwise.

The class is initialized by randomly assigning examples to classes, and updating the statistics for *class_counts* and *feature_sum*.

learnKMeans.py — k-means learning

```
11 from learnProblem import Data_set, Learner, Data_from_file
12 import random
13 import matplotlib.pyplot as plt
14
15 class K_means_learner(Learner):
16
17     def __init__(self,dataset, num_classes):
18         self.dataset = dataset
19         self.num_classes = num_classes
```

```
20          self.random_initialize()
21          self.max_display_level = 5
22
23      def random_initialize(self):
24          # class_counts[c] is the number of examples with class=c
25          self.class_counts = [0]*self.num_classes
26          # feature_sum[f][c] is the sum of the values of feature f for class
                c
27          self.feature_sum = {feat:[0]*self.num_classes
28                              for feat in self.dataset.input_features}
29          for eg in self.dataset.train:
30              cl = random.randrange(self.num_classes) # assign eg to random
                    class
31              self.class_counts[cl] += 1
32              for feat in self.dataset.input_features:
33                  self.feature_sum[feat][cl] += feat(eg)
34          self.num_iterations = 0
35          self.display(1,"Initial class counts: ",self.class_counts)
```

The distance from (the mean of) a class to an example is the sum, over all features, of the sum-of-squares differences of the class mean and the example value.

learnKMeans.py — (continued)

```
37      def distance(self,cl,eg):
38          """distance of the eg from the mean of the class"""
39          return sum( (self.class_prediction(feat,cl)-feat(eg))**2
40                          for feat in self.dataset.input_features)
41
42      def class_prediction(self,feat,cl):
43          """prediction of the class cl on the feature with index feat_ind"""
44          if self.class_counts[cl] == 0:
45              return 0 # arbitrary prediction
46          else:
47              return self.feature_sum[feat][cl]/self.class_counts[cl]
48
49      def class_of_eg(self,eg):
50          """class to which eg is assigned"""
51          return (min((self.distance(cl,eg),cl)
52                          for cl in range(self.num_classes)))[1]
53                # second element of tuple, which is a class with minimum
                      distance
```

One step of k-means updates the *class_counts* and *feature_sum*. It uses the old values to determine the classes, and so the new values for *class_counts* and *feature_sum*. At the end it determines whether the values of these have changes, and then replaces the old ones with the new ones. It returns an indicator of whether the values are stable (have not changed).

learnKMeans.py — (continued)

```
55      def k_means_step(self):
```

```
56          """Updates the model with one step of k-means.
57          Returns whether the assignment is stable.
58          """
59          new_class_counts = [0]*self.num_classes
60          # feature_sum[f][c] is the sum of the values of feature f for class
                c
61          new_feature_sum = {feat: [0]*self.num_classes
62                             for feat in self.dataset.input_features}
63          for eg in self.dataset.train:
64              cl = self.class_of_eg(eg)
65              new_class_counts[cl] += 1
66              for feat in self.dataset.input_features:
67                  new_feature_sum[feat][cl] += feat(eg)
68          stable = (new_class_counts == self.class_counts) and
                (self.feature_sum == new_feature_sum)
69          self.class_counts = new_class_counts
70          self.feature_sum = new_feature_sum
71          self.num_iterations += 1
72          return stable
73
74
75      def learn(self,n=100):
76          """do n steps of k-means, or until convergence"""
77          i=0
78          stable = False
79          while i<n and not stable:
80              stable = self.k_means_step()
81              i += 1
82              self.display(1,"Iteration",self.num_iterations,
83                           "class counts: ",self.class_counts,"
                                 Stable=",stable)
84          return stable
85
86      def show_classes(self):
87          """sorts the data by the class and prints in order.
88          For visualizing small data sets
89          """
90          class_examples = [[] for i in range(self.num_classes)]
91          for eg in self.dataset.train:
92              class_examples[self.class_of_eg(eg)].append(eg)
93          print("Class","Example",sep='\t')
94          for cl in range(self.num_classes):
95              for eg in class_examples[cl]:
96                  print(cl,*eg,sep='\t')
```

Figure 10.3 shows multiple runs for Example 10.5 in Section 10.3.1 of Poole and Mackworth [2023]. Note that the y-axis is sum of squares of the values, which is the square of the Euclidian distance. K-means can stabilize on a different assignment each time it is run. The first run with 2 classes shown in the figure was stable after the first step. The next two runs with 3 classes started

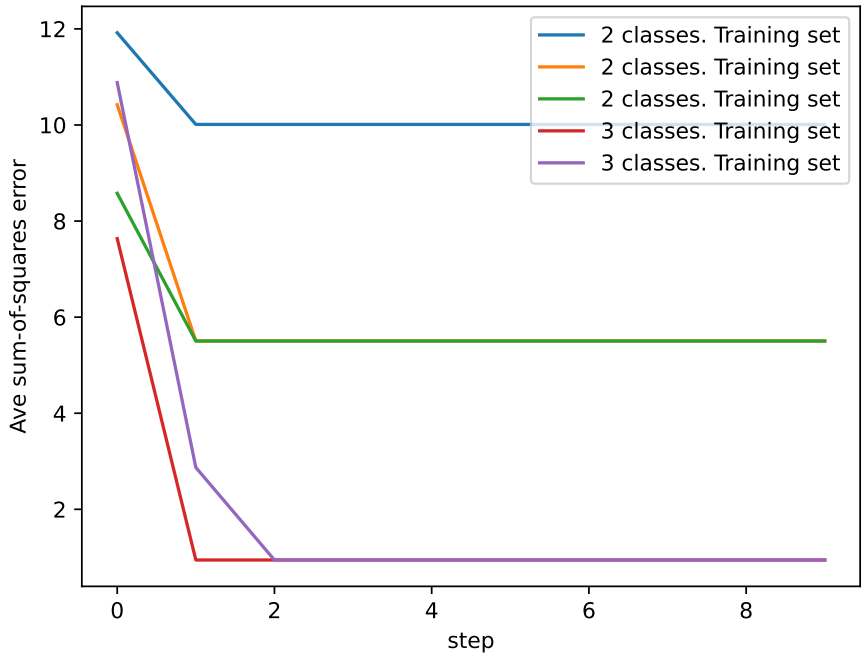


Figure 10.3: k-means plotting error.

with different assignments, but stabilized on the same assignment. (You cannot check if it is the same assignment from the graph, but need to check the assignment of examples to classes.) The second run with 3 classes took tow steps to stabilize, but the other only took one. Note that the algorithm only determines that it is stable with one more run.

learnKMeans.py — (continued)

```
97      def plot_error(self, maxstep=20):
98          """Plots the sum-of-squares error as a function of the number of
                steps"""
99          plt.ion()
100         fig, ax = plt.subplots()
101         ax.set_xlabel("step")
102         ax.set_ylabel("Ave sum-of-squares error")
103         train_errors = []
104         if self.dataset.test:
105             test_errors = []
106         for i in range(maxstep):
107             train_errors.append( sum(self.distance(self.class_of_eg(eg),eg)
108                                      for eg in self.dataset.train)
109                                  /len(self.dataset.train))
110             if self.dataset.test:
```

```
111                 test_errors.append(
                        sum(self.distance(self.class_of_eg(eg),eg)
112                                           for eg in self.dataset.test)
113                                   /len(self.dataset.test))
114             self.learn(1)
115         ax.plot(range(maxstep), train_errors,
116                 label=str(self.num_classes)+" classes. Training set")
117         if self.dataset.test:
118             ax.plot(range(maxstep), test_errors,
119                     label=str(self.num_classes)+" classes. Test set")
120         ax.legend()
121         plt.draw()
122
123 def testKM():
124     # data = Data_from_file('data/emdata1.csv', num_train=10,
            target_index=2000) # trivial example
125     data = Data_from_file('data/emdata2.csv', num_train=10,
            target_index=2000)
126     # data = Data_from_file('data/emdata0.csv', num_train=14,
            target_index=2000) # example from textbook
127     # data = Data_from_file('data/carbool.csv', target_index=2000,
            one_hot=True)
128     kml = K_means_learner(data,2)
129     num_iter=4
130     print("Class assignment after",num_iter,"iterations:")
131     kml.learn(num_iter); kml.show_classes()
132
133 if __name__ == "__main__":
134     testKM()
135
136 # Plot the error
137 # km2=K_means_learner(data,2); km2.plot_error(10) # 2 classes
138 # km3=K_means_learner(data,3); km3.plot_error(10) # 3 classes
139 # km13=K_means_learner(data,10); km13.plot_error(10) # 10 classes
```

Exercise 10.1 If there are many classes, some of the classes can become empty (e.g., try 100 classes with carbool.csv). Implement a way to put some examples into a class, if possible. Two ideas are:

(a) Initialize the classes with actual examples, so that the classes will not start empty. (Do the classes become empty?)

(b) In *class_prediction*, we test whether the code is empty, and make a prediction of 0 for an empty class. It is possible to make a different prediction to "steal" an example (but you should make sure that a class has a consistent value for each feature in a loop).

Make your own suggestions, and compare it with the original, and whichever of these you think may work better.

10.3 EM

In the following definition, a class, c, is a integer in range $[0, num\_classes)$. i is an index of a feature, so $feat[i]$ is the ith feature, and a feature is a function from tuples to values. val is a value of a feature.

A model consists of 2 lists, which form the sufficient statistics:

- $class\_counts$ is a list such that $class\_counts[c]$ is the number of tuples with $class = c$, where each tuple is weighted by its probability, i.e.,

$$class\_counts[c] = \sum_{t:class(t)=c} P(t)$$

- $feature\_counts$ is a list such that $feature\_counts[i][val][c]$ is the weighted count of the number of tuples t with $feat[i](t) = val$ and $class(t) = c$, each tuple is weighted by its probability, i.e.,

$$feature\_counts[i][val][c] = \sum_{t:feat[i](t)=val \text{ and } class(t)=c} P(t)$$

learnEM.py — EM Learning

```
from learnProblem import Data_set, Learner, Data_from_file
import random
import math
import matplotlib.pyplot as plt

class EM_learner(Learner):
    def __init__(self,dataset, num_classes):
        self.dataset = dataset
        self.num_classes = num_classes
        self.class_counts = None
        self.feature_counts = None
```

The function *em_step* goes though the training examples, and updates these counts. The first time it is run, when there is no model, it uses random distributions.

learnEM.py — (continued)

```
    def em_step(self, orig_class_counts, orig_feature_counts):
        """updates the model."""
        class_counts = [0]*self.num_classes
        feature_counts = [{val:[0]*self.num_classes
                           for val in feat.frange}
                           for feat in self.dataset.input_features]
        for tple in self.dataset.train:
            if orig_class_counts: # a model exists
                tpl_class_dist = self.prob(tple, orig_class_counts,
                    orig_feature_counts)
```

```
32          else:               # initially, with no model, return a random
                distribution
33              tpl_class_dist = random_dist(self.num_classes)
34          for cl in range(self.num_classes):
35              class_counts[cl] += tpl_class_dist[cl]
36              for (ind,feat) in enumerate(self.dataset.input_features):
37                  feature_counts[ind][feat(tple)][cl] += tpl_class_dist[cl]
38      return class_counts, feature_counts
```

prob computes the probability of a class c for a tuple *tpl*, given the current statistics.

$$\begin{aligned} P(c \mid tple) &\propto P(c) * \prod_i P(X_i{=}tple(i) \mid c) \\ &= \frac{class\_counts[c]}{len(self.dataset)} * \prod_i \frac{feature\_counts[i][feat_i(tple)][c]}{class\_counts[c]} \\ &\propto \frac{\prod_i feature\_counts[i][feat_i(tple)][c]}{class\_counts[c]^{|feats|-1}} \end{aligned}$$

The last step is because $len(self.dataset)$ is a constant (independent of c). $class\_counts[c]$ can be taken out of the product, but needs to be raised to the power of the number of features, and one of them cancels.

learnEM.py — (continued)

```
40  def prob(self, tple, class_counts, feature_counts):
41      """returns a distribution over the classes for tuple tple in the
            model defined by the counts
42      """
43      feats = self.dataset.input_features
44      unnorm = [prod(feature_counts[i][feat(tple)][c]
45                     for (i,feat) in enumerate(feats))
46                /(class_counts[c]**(len(feats)-1))
47                for c in range(self.num_classes)]
48      thesum = sum(unnorm)
49      return [un/thesum for un in unnorm]
```

learn does n steps of EM:

learnEM.py — (continued)

```
51  def learn(self,n):
52      """do n steps of em"""
53      for i in range(n):
54          self.class_counts,self.feature_counts =
                self.em_step(self.class_counts,
55                                                  self.feature_counts)
```

The following is for visualizing the classes. It prints the dataset ordered by the probability of class c.

learnEM.py — (continued)

```
57  def show_class(self,c):
```

```
58        """sorts the data by the class and prints in order.
59        For visualizing small data sets
60        """
61        sorted_data =
              sorted((self.prob(tpl,self.class_counts,self.feature_counts)[c],
62                            ind,  # preserve ordering for equal
                                  probabilities
63                            tpl)
64                           for (ind,tpl) in enumerate(self.dataset.train))
65        for cc,r,tpl in sorted_data:
66            print(cc,*tpl,sep='\t')
```

The following are for evaluating the classes.

The probability of a tuple can be evaluated by marginalizing over the classes:

$$\begin{aligned} P(tple) &= \sum_c P(c) * \prod_i P(X_i{=}tple(i) \mid c) \\ &= \sum_c \frac{cc[c]}{len(self.dataset)} * \prod_i \frac{fc[i][feat_i(tple)][c]}{cc[c]} \end{aligned}$$

where cc is the class count and fc is feature count. $len(self.dataset)$ can be distributed out of the sum, and $cc[c]$ can be taken out of the product:

$$= \frac{1}{len(self.dataset)} \sum_c \frac{1}{cc[c]^{\#feats-1}} * \prod_i fc[i][feat_i(tple)][c]$$

Given the probability of each tuple, we can evaluate the logloss, as the negative of the log probability:

learnEM.py — (continued)

```
68    def logloss(self,tple):
69        """returns the logloss of the prediction on tple, which is
              -log(P(tple))
70        based on the current class counts and feature counts
71        """
72        feats = self.dataset.input_features
73        res = 0
74        cc = self.class_counts
75        fc = self.feature_counts
76        for c in range(self.num_classes):
77            res += prod(fc[i][feat(tple)][c]
78                        for (i,feat) in
                                enumerate(feats))/(cc[c]**(len(feats)-1))
79        if res>0:
80            return -math.log2(res/len(self.dataset.train))
81        else:
82            return float("inf") #infinity
```

Figure 10.4 shows the training and test error for various numbers of classes for the carbool dataset (calls commented out at the end of the code).

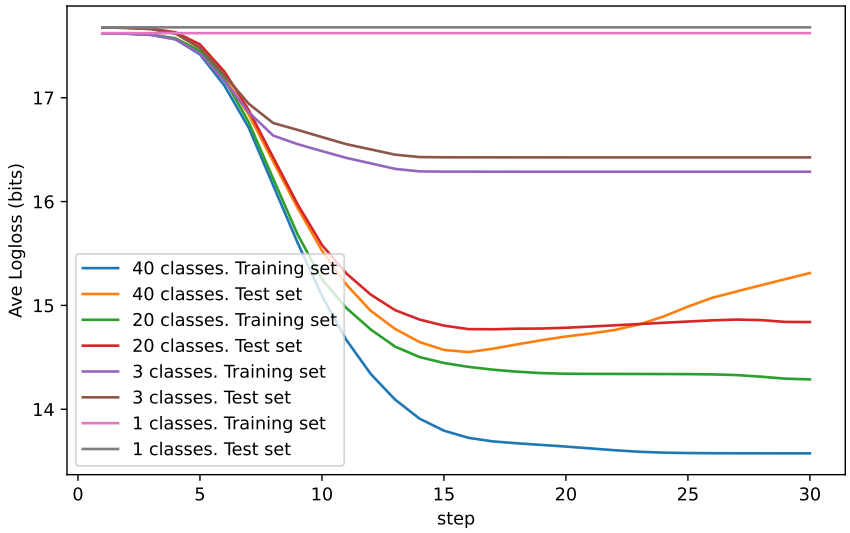


Figure 10.4: EM plotting error.

learnEM.py — (continued)

```
84      def plot_error(self, maxstep=20):
85          """Plots the logloss error as a function of the number of steps"""
86          plt.ion()
87          ax.set_xlabel("step")
88          ax.set_ylabel("Ave Logloss (bits)")
89          train_errors = []
90          if self.dataset.test:
91              test_errors = []
92          for i in range(maxstep):
93              self.learn(1)
94              train_errors.append( sum(self.logloss(tple) for tple in
                    self.dataset.train)
95                                  /len(self.dataset.train))
96              if self.dataset.test:
97                  test_errors.append( sum(self.logloss(tple) for tple in
                        self.dataset.test)
98                                      /len(self.dataset.test))
99          ax.plot(range(1,maxstep+1),train_errors,
100                  label=str(self.num_classes)+" classes. Training set")
101         if self.dataset.test:
102             ax.plot(range(1,maxstep+1),test_errors,
103                     label=str(self.num_classes)+" classes. Test set")
104         ax.legend()
105         plt.show()
106
107 # global variables so the plots can share axes.
```

```
fig, ax = plt.subplots()

def prod(L):
    """returns the product of the elements of L"""
    res = 1
    for e in L:
        res *= e
    return res

def random_dist(k):
    """generate k random numbers that sum to 1"""
    res = [random.random() for i in range(k)]
    s = sum(res)
    return [v/s for v in res]

def testEM():
    print("testing EM")
    global data, eml
    data = Data_from_file('data/emdata2.csv', num_train=10,
        target_index=2000)
    # data = Data_from_file('data/carbool.csv', target_index=2000,
        one_hot=True)
    eml = EM_learner(data,2)
    num_iter=2
    print("Class assignment after",num_iter,"iterations:")
    eml.learn(num_iter); eml.show_class(0)

if __name__ == "__main__":
    testEM()

 # Plot the error
 # em1=EM_learner(data,1); em1.plot_error(30) # 1 class (predict mean)
# em2=EM_learner(data,2); em2.plot_error(40) # 2 classes
# em3=EM_learner(data,3); em3.plot_error(40) # 3 classes
# em10=EM_learner(data,10); em10.plot_error(40) # 10 classes
# em13=EM_learner(data,13); em13.plot_error(40) # 13 classes

# show the values for the variables
# [f.frange for f in data.input_features]
```

Exercise 10.2 For data where there are naturally 2 classes, does EM with 3 classes do better on the training set after a while than 2 classes? Is is better on a test set. Explain why. Hint: look what the 3 classes are. Use "eml.show_class(i)" for each of the classes $i \in [0,3)$.

Exercise 10.3 Write code to plot the logloss as a function of the number of classes (from 1 to, say, 30) for a fixed number of iterations. (From the experience with the existing code, think about how many iterations are appropriate.

Exercise 10.4 Repeat the previous exercise, but use cross validation to select the number of iterations as a function of the number of classes and other features of

the dataset.

Chapter 11

Causality

11.1 Do Questions

A causal model can answer "do" questions.

The intervene function takes a belief network and a *variable* : *value* dictionary specifying what to "do", and returns a belief network resulting from intervening to set each variable in the dictionary to its value specified. It replaces the conditional probability distribution, CPD, (Section 9.3) of each intervened variable with an constant CPD.

probDo.py — Probabilistic inference with the do operator

```
from probGraphicalModels import InferenceMethod, BeliefNetwork
from probFactors import CPD, ConstantCPD

def intervene(bn, do={}):
    assert isinstance(bn, BeliefNetwork), f"Do only applies to belief
        networks ({bn.title})"
    if do=={}:
        return bn
    else:
        newfacs = ({f for (ch,f) in bn.var2cpt.items() if ch not in do} |
                   {ConstantCPD(v,c) for (v,c) in do.items()})
        return BeliefNetwork(f"{bn.title}(do={do})", bn.variables, newfacs)
```

The following adds the queryDo method to the InferenceMethod class, so it can be used with any inference method. It replaces the graphical model with the modified one, runs the inference algorithm, and restores the initial belief network.

probDo.py — (continued)

```
def queryDo(self, qvar, obs={}, do={}):
```

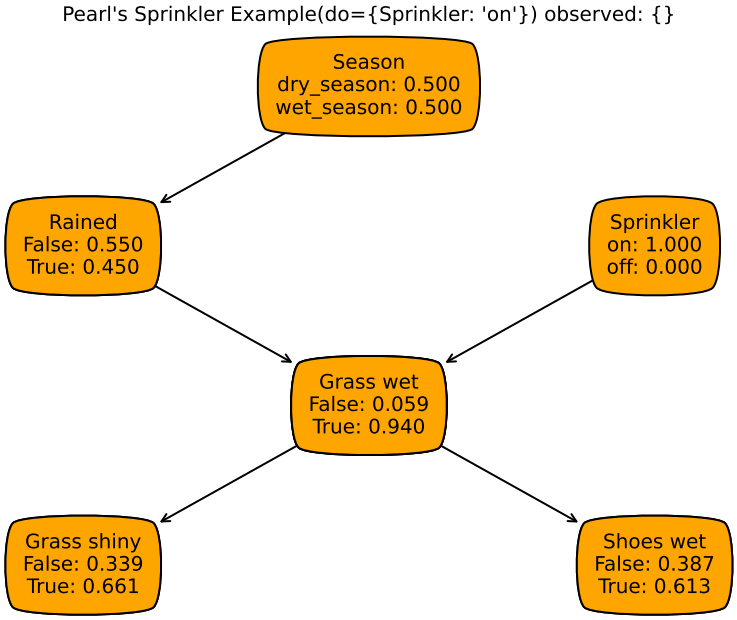


Figure 11.1: The sprinkler belief network with do={Sprinkler:"on"}.

```
    """Extends query method to also allow for interventions.
    """
    oldBN, self.gm = self.gm, intervene(self.gm, do)
    result = self.query(qvar, obs)
    self.gm = oldBN # restore original
    return result

# make queryDo available for all inference methods
InferenceMethod.queryDo = queryDo
```

The following example is based on the sprinkler belief network of Section 9.4.2 shown in Figure 9.4. The network with the intervention of putting the sprinkler on is shown in Figure 11.1.

probDo.py — (continued)

```
from probRC import ProbRC

from probExamples import bn_sprinkler, Season, Sprinkler, Rained,
    Grass_wet, Grass_shiny, Shoes_wet
bn_sprinklerv = ProbRC(bn_sprinkler)
## bn_sprinklerv.queryDo(Shoes_wet)
## bn_sprinklerv.queryDo(Shoes_wet,obs={Sprinkler:"on"})
## bn_sprinklerv.queryDo(Shoes_wet,do={Sprinkler:"on"})
## bn_sprinklerv.queryDo(Season, obs={Sprinkler:"on"})
## bn_sprinklerv.queryDo(Season, do={Sprinkler:"on"})
```

Figure 11.2: Does taking marijuana lead to hard drugs: observable variables

```
43
44 ### Showing posterior distributions:
45 # bn_sprinklerv.show_post({})
46 # bn_sprinklerv.show_post({Sprinkler:"on"})
47 # spon = intervene(bn_sprinkler, do={Sprinkler:"on"})
48 # ProbRC(spon).show_post({})
```

The following is a representation of a possible model where marijuana is a gateway drug to harder drugs (or not). Before reading the code, try the commented-out queries at the end. Figure 11.2 shows the network with the observable variables, `Takes_Marijuana` and `Takes_Hard_Drugs`.

probDo.py — (continued)

```
50 from variable import Variable
51 from probFactors import Prob
52 from probGraphicalModels import BeliefNetwork
53 boolean = [False, True]
54
55 Drug_Prone = Variable("Drug_Prone", boolean, position=(0.1,0.5)) #
       (0.5,0.9))
56 Side_Effects = Variable("Side_Effects", boolean, position=(0.1,0.5)) #
       (0.5,0.1))
57 Takes_Marijuana = Variable("\nTakes_Marijuana\n", boolean,
       position=(0.1,0.5))
58 Takes_Hard_Drugs = Variable("Takes_Hard_Drugs", boolean,
       position=(0.9,0.5))
59
60 p_dp = Prob(Drug_Prone, [], [0.8, 0.2])
61 p_be = Prob(Side_Effects, [Takes_Marijuana], [[1, 0], [0.4, 0.6]])
62 p_tm = Prob(Takes_Marijuana, [Drug_Prone], [[0.98, 0.02], [0.2, 0.8]])
63 p_thd = Prob(Takes_Hard_Drugs, [Side_Effects, Drug_Prone],
64                # Drug_Prone=False  Drug_Prone=True
65                [[[0.999, 0.001],   [0.6, 0.4]], # Side_Effects=False
66                 [[0.99999, 0.00001], [0.995, 0.005]]]) # Side_Effects=True
67
```

```
drugs = BeliefNetwork("Gateway Drug?",
                  [Drug_Prone,Side_Effects, Takes_Marijuana,
                      Takes_Hard_Drugs],
                  [p_tm, p_dp, p_be, p_thd])

drugsq = ProbRC(drugs)
# drugsq.queryDo(Takes_Hard_Drugs)
# drugsq.queryDo(Takes_Hard_Drugs, obs = {Takes_Marijuana: True})
# drugsq.queryDo(Takes_Hard_Drugs, obs = {Takes_Marijuana: False})
# drugsq.queryDo(Takes_Hard_Drugs, do = {Takes_Marijuana: True})
# drugsq.queryDo(Takes_Hard_Drugs, do = {Takes_Marijuana: False})

# ProbRC(drugs).show_post({})
# ProbRC(drugs).show_post({Takes_Marijuana: True})
# ProbRC(drugs).show_post({Takes_Marijuana: False})
# ProbRC(intervene(drugs, do={Takes_Marijuana: True})).show_post({})
# ProbRC(intervene(drugs, do={Takes_Marijuana: False})).show_post({})
# Why was that? Try the following then repeat:
# Drug_Prone.position=(0.5,0.9); Side_Effects.position=(0.5,0.1)
```

11.2 Counterfactual Reasoning

The following provides two examples of counterfactual reasoning. In the following code, the user has to provide the deterministic system with noise. As we will see, there are multiple deterministic systems with noise that can produce the same causal probabilities.

probCounterfactual.py — Counterfactual Query Example

```
from variable import Variable
from probFactors import Prob, ProbDT, IFeq, SameAs, Dist
from probGraphicalModels import BeliefNetwork
from probRC import ProbRC
from probDo import queryDo

boolean = [False, True]
```

11.2.1 Choosing Deterministic System

This section presents an example to encourage you to think about what deterministic system to use.

Consider the following example (thanks to Sophie Song). Suppose Bob went on a date with Alice. Bob was either on time or not (variable B is true when Bob is on time). Alice, who is fastidious about punctuality chooses whether to go on a second date (variable A is true when Alice agrees to a second date). Whether Bob is late depends on which cab company he called (variable C). Suppose Bob calls one of the cab companies, he was late, and Alice doesn't ask for a second date. Bob wonders "what if I had called the other

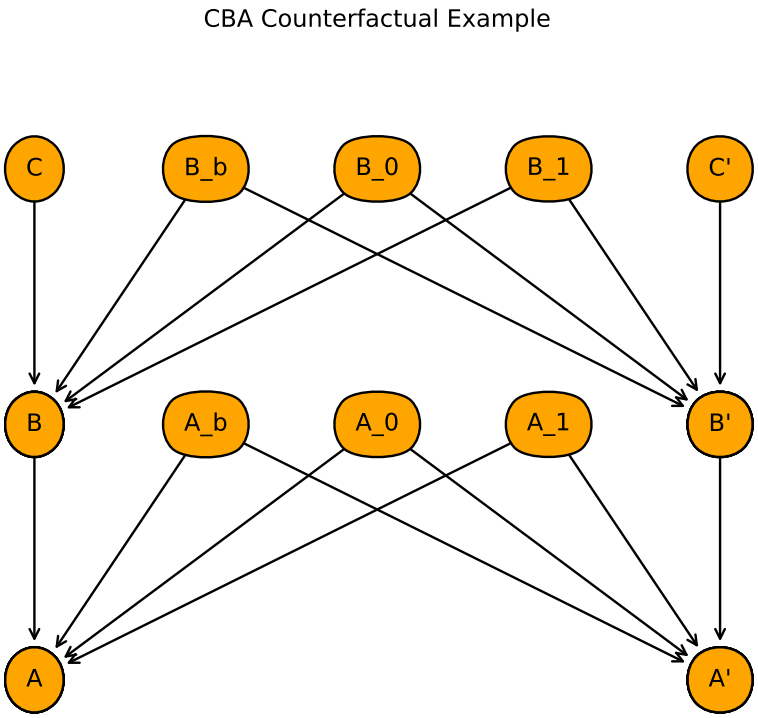


Figure 11.3: $C \rightarrow B \rightarrow A$ belief network for "what if C". Figure generated by by `cbaCounter.show()`

cab company". Suppose all variables are Boolean. C causally depends on B, and not directly on C, and B depends on C, so the appropriate causal model is $C \rightarrow B \rightarrow A$.

Assume the following probabilities obtained from observations (where the lower case c represents $C = true$, and similarly for other variables):

$$P(c) = 0.5$$
$$P(b \mid c) = P(b \mid \neg c) = 0.7 \quad \text{(the cab companies are equally reliable)}$$
$$(a \mid b) = 0.4, (a \mid \neg b) = 0.2.$$

Consider "what if C was True" or "what if C was False". For example, suppose A=*false* and C=*false* is observed and you want the probability of A if C were false.

Figure 11.3 shows the paired network for "what if C". The primed variables represent the situation where C is counterfactually True or False. In this network, `Cprime` should be conditioned on. Conditioning on `Cprime` should not affect the non-primed variables. (You should check this).

probCounterfactual.py — (continued)

```
# as a deterministic system with independent noise
C = Variable("C", boolean, position=(0.1,0.8))
B = Variable("B", boolean, position=(0.1,0.4))
A = Variable("A", boolean, position=(0.1,0.0))
Cprime = Variable("C'", boolean, position=(0.9,0.8))
Bprime = Variable("B'", boolean, position=(0.9,0.4))
Aprime = Variable("A'", boolean, position=(0.9,0.0))
B_b = Variable("B_b", boolean, position=(0.3,0.8))
B_0 = Variable("B_0", boolean, position=(0.5,0.8))
B_1 = Variable("B_1", boolean, position=(0.7,0.8))
A_b = Variable("A_b", boolean, position=(0.3,0.4))
A_0 = Variable("A_0", boolean, position=(0.5,0.4))
A_1 = Variable("A_1", boolean, position=(0.7,0.4))
```

The conditional probability $P(A \mid B)$ is represented using three noise parameters, A_b, A_0 and A_1, with the equivalence:

$$a \equiv a_b \vee (\neg b \wedge a_0) \vee (b \wedge a_1)$$

Thus a_b is the background cause of a, a_0 is the cause used when B=*false* and a_1 is the cause used when B=*false*. Note that this is over parametrized with respect the belief network, using three parameters whereas arbitrary conditional probability can be represented using two parameters.

The running example where $(a \mid b) = 0.4$ and $(a \mid \neg b) = 0.2$ can be represented using

$$P(a_b) = 0, P(a_0) = 0.2, P(a_1) = 0.4$$

or

$$P(a_b) = 0.2, P(a_0) = 0, P(a_1) = 0.25$$

(and infinitely many others between these). These cannot be distinguished by observations or by interventions. As you can see if you play with the code, these have different counterfactual conclusions.

$P(B \mid C)$ is represented similarly, using variables B_b, B_0, and B_1.

The following code uses the decision tree representation of conditional probabilities of Section 9.3.4.

probCounterfactual.py — (continued)

```
p_C = Prob(C, [], [0.5,0.5])
p_B = ProbDT(B, [C, B_b, B_0, B_1], IFeq(B_b,True,Dist([0,1]),
                                         IFeq(C,True,SameAs(B_1),SameAs(B_0))))
p_A = ProbDT(A, [B, A_b, A_0, A_1], IFeq(A_b,True,Dist([0,1]),
                                         IFeq(B,True,SameAs(A_1),SameAs(A_0))))
p_Cprime = Prob(Cprime,[], [0.5,0.5])
p_Bprime = ProbDT(Bprime, [Cprime, B_b, B_0, B_1],
                    IFeq(B_b,True,Dist([0,1]),
```

```
                                  IFeq(Cprime,True,SameAs(B_1),SameAs(B_0))))
p_Aprime = ProbDT(Aprime, [Bprime, A_b, A_0, A_1],
                    IFeq(A_b,True,Dist([0,1]),
                                  IFeq(Bprime,True,SameAs(A_1),SameAs(A_0))))
p_b_b = Prob(B_b, [], [1,0])
p_b_0 = Prob(B_0, [], [0.3,0.7])
p_b_1 = Prob(B_1, [], [0.3,0.7])

p_a_b = Prob(A_b, [], [1,0])
p_a_0 = Prob(A_0, [], [0.8,0.2])
p_a_1 = Prob(A_1, [], [0.6,0.4])

p_b_np = Prob(B, [], [0.3,0.7]) # for AB network
p_Bprime_np = Prob(Bprime, [], [0.3,0.7]) # for AB network
ab_Counter = BeliefNetwork("AB Counterfactual Example",
                   [A,B,Aprime,Bprime, A_b,A_0,A_1],
                   [p_A, p_b_np, p_Aprime, p_Bprime_np, p_a_b, p_a_0,
                       p_a_1])

cbaCounter = BeliefNetwork("CBA Counterfactual Example",
                   [A,B,C, Aprime,Bprime,Cprime, B_b,B_0,B_1, A_b,A_0,A_1],
                   [p_A, p_B, p_C, p_Aprime, p_Bprime, p_Cprime,
                        p_b_b, p_b_0, p_b_1, p_a_b, p_a_0, p_a_1])
```

Here are some queries you might like to try. The show_post queries might be most useful if you have the space to show multiple queries.

probCounterfactual.py — (continued)

```
cbaq = ProbRC(cbaCounter)
# cbaq.queryDo(Aprime, obs = {C:True, Cprime:False})
# cbaq.queryDo(Aprime, obs = {C:False, Cprime:True})
# cbaq.queryDo(Aprime, obs = {A:True, C:True, Cprime:False})
# cbaq.queryDo(Aprime, obs = {A:False, C:True, Cprime:False})
# cbaq.queryDo(Aprime, obs = {A:False, C:True, Cprime:False})
# cbaq.queryDo(A_1, obs = {C:True,Aprime:False})
# cbaq.queryDo(A_0, obs = {C:True,Aprime:False})

# cbaq.show_post(obs = {})
# cbaq.show_post(obs = {C:True, Cprime:False})
# cbaq.show_post(obs = {A:False, C:True, Cprime:False})
# cbaq.show_post(obs = {A:True, C:True, Cprime:False})
```

Exercise 11.1 Consider the scenario "Bob called the first cab ($C = true$), was late and Alice agrees to a second date". What would you expect from the scenario "what if Bob called the other cab?". What does the network predict? Design probabilities for the noise variables that fits the conditional probability and also fits your expectation.

Exercise 11.2 How would you expect the counterfactual conclusion to change given the following two scenarios that fit the story:

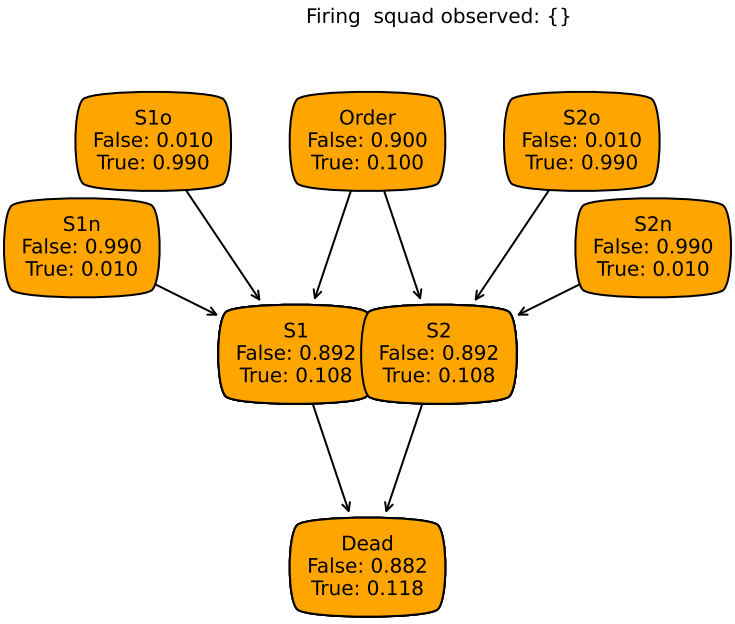


Figure 11.4: Firing squad belief network (figure obtained from `fsq.show_post({})`

- The cabs are both very reliable and start at the same location (and so face the same traffic).
- The cabs are each 90% reliable and start from opposite directions.

(a) How would you expect the predictions to differ in these two cases?

(b) How can you fit the conditional probabilities above and represent each of these by changing the probabilities of the noise variables?

(c) How can these be learned from data? (Hint: consider learning a correlation between the taxi arrivals). Is your approach always applicable? If not, for which cases is it applicable or not.

Exercise 11.3 Choose two assignments to values to each of a_b, a_0 and a_1 using $a \equiv a_b \vee (\neg b \wedge a_0) \vee (b \wedge a_1)$, and a counterfactual query such that (a) the two assignments cannot be distinguished by observations or by interventions, and (b) the predictions for the query differ by an arbitrarluy large amount (differ by $1 - \epsilon$ for a small value of ϵ, such as $\epsilon = 0.1$).

11.2.2 Firing Squad Example

The following is the firing squad example of Pearl [2009] as a deterministic system. See Figure 11.4.

probCounterfactual.py — (continued)

```
Order = Variable("Order", boolean, position=(0.4,0.8))
S1 = Variable("S1", boolean, position=(0.3,0.4))
S1o = Variable("S1o", boolean, position=(0.1,0.8))
S1n = Variable("S1n", boolean, position=(0.0,0.6))
S2 = Variable("S2", boolean, position=(0.5,0.4))
S2o = Variable("S2o", boolean, position=(0.7,0.8))
S2n = Variable("S2n", boolean, position=(0.8,0.6))
Dead = Variable("Dead", boolean, position=(0.4,0.0))
```

Instead of the tabular representation of the if-then-else structure used for the $A \rightarrow B \rightarrow C$ network above, the following uses the decision tree representation of conditional probabilities of Section 9.3.4.

probCounterfactual.py — (continued)

```
p_S1 = ProbDT(S1, [Order, S1o, S1n],
                  IFeq(Order,True, SameAs(S1o), SameAs(S1n)))
p_S2 = ProbDT(S2, [Order, S2o, S2n],
                  IFeq(Order,True, SameAs(S2o), SameAs(S2n)))
p_dead = Prob(Dead, [S1,S2], [[[1,0],[0,1]],[[0,1],[0,1]]])
                #IFeq(S1,True,True,SameAs(S2)))
p_order = Prob(Order, [], [0.9, 0.1])
p_s1o = Prob(S1o, [], [0.01, 0.99])
p_s1n = Prob(S1n, [], [0.99, 0.01])
p_s2o = Prob(S2o, [], [0.01, 0.99])
p_s2n = Prob(S2n, [], [0.99, 0.01])

firing_squad = BeliefNetwork("Firing squad",
                         [Order, S1, S1o, S1n, S2, S2o, S2n, Dead],
                         [p_order, p_dead, p_S1, p_s1o, p_s1n, p_S2, p_s2o,
                             p_s2n])
fsq = ProbRC(firing_squad)
# fsq.queryDo(Dead)
# fsq.queryDo(Order, obs={Dead:True})
# fsq.queryDo(Dead, obs={Order:True})
# fsq.show_post({})
# fsq.show_post({Dead:True})
# fsq.show_post({S2:True})
```

Exercise 11.4 Create the network for "what if shooter 2 did or did not shoot". Give the probabilities of the following counterfactuals:

(a) The prisoner is dead; what is the probability that the prisoner would be dead if shooter 2 did not shoot?

(b) Shooter 2 shot; what is the probability that the prisoner would be dead if shooter 2 did not shoot?

(c) No order was given, but the prisoner is dead; what is the probability that the prisoner would be dead if shooter 2 did not shoot?

Exercise 11.5 Create the network for "what if the order was or was not given". Give the probabilities of the following counterfactuals:

(a) The prisoner is dead; what is the probability that the prisoner would be dead if the order was not given?

(b) The prisoner is not dead; what is the probability that the prisoner would be dead if the order was not given? (Is this different from the prior that the prisoner is dead, or the posterior that the prisoner was dead given the order was not given).

(c) Shooter 2 shot; what is the probability that the prisoner would be dead if the order was not given?

(d) Shooter 2 did not shoot; what is the probability that the prisoner would be dead if the order was given? (Is this different from the probability that the the prisoner would be dead if the order was given without the counterfactual observation)?

Chapter 12

Planning with Uncertainty

12.1 Decision Networks

The decision network code builds on the representation for belief networks of Chapter 9.

First, define factors that define the utility. Here the **utility** is a function of the variables in *vars*. In a **utility table** the utility is defined in terms of a tabular factor – a list that enumerates the values – as in Section 9.3.3. Another representations for factors (Section 9.2) could able be used.

decnNetworks.py — Representations for Decision Networks

```
from probGraphicalModels import GraphicalModel, BeliefNetwork
from probFactors import Factor, CPD, TabFactor, factor_times, Prob
from variable import Variable
import matplotlib.pyplot as plt

class Utility(Factor):
    """A factor defining a utility"""
    pass

class UtilityTable(TabFactor, Utility):
    """A factor defining a utility using a table"""
    def __init__(self, vars, table, position=None):
        """Creates a factor on vars from the table.
        The table is ordered according to vars.
        """
        TabFactor.__init__(self,vars,table, name="Utility")
        self.position = position
```

A **decision variable** is like a random variable with a string name, and a domain, which is a list of possible values. The decision variable also includes the

parents, a list of the variables whose value will be known when the decision is made. It also includes a position, which is used for plotting.

decnNetworks.py — (continued)

```
class DecisionVariable(Variable):
    def __init__(self, name, domain, parents, position=None):
        Variable.__init__(self, name, domain, position)
        self.parents = parents
        self.all_vars = set(parents) | {self}
```

A decision network is a graphical model where the variables can be random variables or decision variables. Among the factors we assume there is one utility factor. Note that this is an instance of `BeliefNetwork` but overrides `__init__`.

decnNetworks.py — (continued)

```
class DecisionNetwork(BeliefNetwork):
    def __init__(self, title, vars, factors):
        """title is a string
        vars is a list of variables (random and decision)
        factors is a list of factors (instances of CPD and Utility)
        """
        GraphicalModel.__init__(self, title, vars, factors)
              # not BeliefNetwork.__init__
        self.var2parents = ({v : v.parents for v in vars
                                if isinstance(v,DecisionVariable)}
                    | {f.child:f.parents for f in factors
                           if isinstance(f,CPD)})
        self.children = {n:[] for n in self.variables}
        for v in self.var2parents:
            for par in self.var2parents[v]:
                self.children[par].append(v)
        self.utility_factor = [f for f in factors
                                  if isinstance(f,Utility)][0]
        self.topological_sort_saved = None

    def __str__(self):
        return self.title
```

The split order ensures that the parents of a decision node are split before the decision node, and no other variables (if that is possible).

decnNetworks.py — (continued)

```
    def split_order(self):
        so = []
        tops = self.topological_sort()
        for v in tops:
            if isinstance(v,DecisionVariable):
                so += [p for p in v.parents if p not in so]
                so.append(v)
        so += [v for v in tops if v not in so]
        return so
```

decnNetworks.py — (continued)

```
    def show(self, fontsize=10,
            colors={'utility':'red', 'decision':'lime', 'random':'orange'}):
        plt.ion()  # interactive
        fig, ax = plt.subplots()
        ax.set_axis_off()
        ax.set_title(self.title, fontsize=fontsize)
        for par in self.utility_factor.variables:
            ax.annotate("Utility", par.position,
                        xytext=self.utility_factor.position,
                        arrowprops={'arrowstyle':'<-'},
                        bbox=dict(boxstyle="sawtooth,pad=1.0",
                                  facecolor=colors['utility']),
                        ha='center', va='center', fontsize=fontsize)
        for var in reversed(self.topological_sort()):
            if isinstance(var,DecisionVariable):
                bbox = dict(boxstyle="square,pad=1.0",
                            facecolor=colors['decision'])
            else:
               bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
                           facecolor=colors['random'])
            if self.var2parents[var]:
                for par in self.var2parents[var]:
                    ax.annotate(var.name, par.position, xytext=var.position,
                                arrowprops={'arrowstyle':'<-'},bbox=bbox,
                                ha='center', va='center',
                                fontsize=fontsize)
            else:
                x,y = var.position
                ax.text(x,y,var.name,bbox=bbox,ha='center', va='center',
                    fontsize=fontsize)
```

12.1.1 Example Decision Networks

Umbrella Decision Network

Here is a simple "umbrella" decision network. The output of `umbrella_dn.show()` is shown in Figure 12.1.

decnNetworks.py — (continued)

```
Weather = Variable("Weather", ["NoRain", "Rain"],
                   position=(0.5,0.8))
Forecast = Variable("Forecast", ["Sunny", "Cloudy", "Rainy"],
                    position=(0,0.4))
# Each variant uses one of the following:
Umbrella = DecisionVariable("Umbrella", ["Take", "Leave"], {Forecast},
                            position=(0.5,0))

p_weather = Prob(Weather, [], {"NoRain":0.7, "Rain":0.3})
p_forecast = Prob(Forecast, [Weather],
```

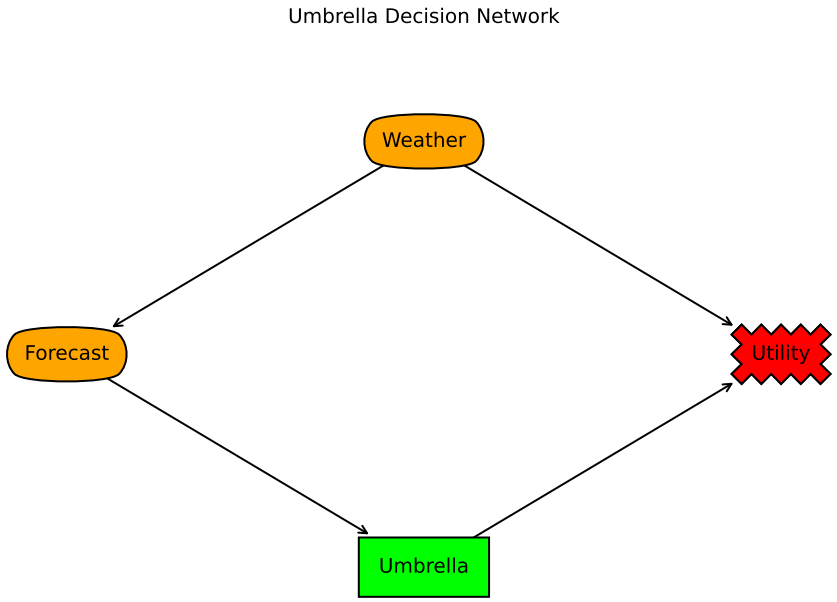


Figure 12.1: The umbrella decision network. Figure generated by `umbrella_dn.show()`

```
                    {"NoRain":{"Sunny":0.7, "Cloudy":0.2, "Rainy":0.1},
                     "Rain":{"Sunny":0.15, "Cloudy":0.25, "Rainy":0.6}})
umb_utility = UtilityTable([Weather, Umbrella],
                      {"NoRain":{"Take":20, "Leave":100},
                       "Rain":{"Take":70, "Leave":0}}, position=(1,0.4))

umbrella_dn = DecisionNetwork("Umbrella Decision Network",
                              {Weather, Forecast, Umbrella},
                              {p_weather, p_forecast, umb_utility})

# umbrella_dn.show()
# umbrella_dn.show(fontsize=15)
```

The following is a variant with the umbrella decision having 2 parents; nothing else has changed. This is interesting because one of the parents is not needed; if the agent knows the weather, it can ignore the forecast.

decnNetworks.py — (continued)

```
Umbrella2p = DecisionVariable("Umbrella", ["Take", "Leave"],
                              {Forecast, Weather}, position=(0.5,0))
umb_utility2p = UtilityTable([Weather, Umbrella2p],
                             {"NoRain":{"Take":20, "Leave":100},
                              "Rain":{"Take":70, "Leave":0}},
                             position=(1,0.4))
```

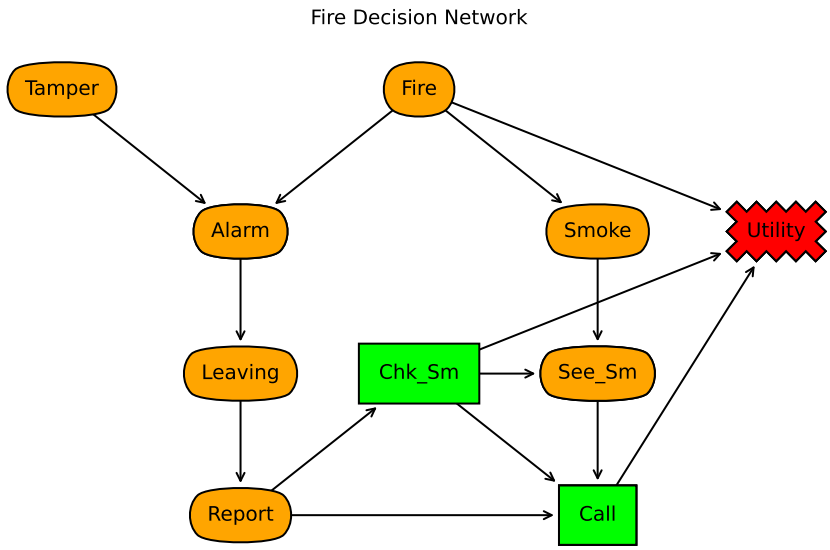


Figure 12.2: Fire Decision Network. Figure generated by `fire_dn.show()`

```
umbrella_dn2p = DecisionNetwork("Umbrella Decision Network (extra arc)",
                                {Weather, Forecast, Umbrella2p},
                                {p_weather, p_forecast, umb_utility2p})

# umbrella_dn2p.show()
# umbrella_dn2p.show(fontsize=15)
```

Fire Decision Network

The fire decision network of Figure 12.2 (showing the result of `fire_dn.show()`) is represented as:

decnNetworks.py — (continued)

```
boolean = [False, True]
Alarm = Variable("Alarm", boolean, position=(0.25,0.633))
Fire = Variable("Fire", boolean, position=(0.5,0.9))
Leaving = Variable("Leaving", boolean, position=(0.25,0.366))
Report = Variable("Report", boolean, position=(0.25,0.1))
Smoke = Variable("Smoke", boolean, position=(0.75,0.633))
Tamper = Variable("Tamper", boolean, position=(0,0.9))

See_Sm = Variable("See_Sm", boolean, position=(0.75,0.366) )
Chk_Sm = DecisionVariable("Chk_Sm", boolean, {Report},
```

```
                            position=(0.5, 0.366))
Call = DecisionVariable("Call", boolean,{See_Sm,Chk_Sm,Report},
                          position=(0.75,0.1))

f_ta = Prob(Tamper,[],[0.98,0.02])
f_fi = Prob(Fire,[],[0.99,0.01])
f_sm = Prob(Smoke,[Fire],[[0.99,0.01],[0.1,0.9]])
f_al = Prob(Alarm,[Fire,Tamper],[[[0.9999, 0.0001], [0.15, 0.85]],
                                 [[0.01, 0.99], [0.5, 0.5]]])
f_lv = Prob(Leaving,[Alarm],[[0.999, 0.001], [0.12, 0.88]])
f_re = Prob(Report,[Leaving],[[0.99, 0.01], [0.25, 0.75]])
f_ss = Prob(See_Sm,[Chk_Sm,Smoke],[[[1,0],[1,0]],[[1,0],[0,1]]])

ut = UtilityTable([Chk_Sm,Fire,Call],
                  [[[0,-200],[-5000,-200]],[[-20,-220],[-5020,-220]]],
                  position=(1,0.633))

fire_dn = DecisionNetwork("Fire Decision Network",
                      {Tamper,Fire,Alarm,Leaving,Smoke,Call,See_Sm,Chk_Sm,Report},
                      {f_ta,f_fi,f_sm,f_al,f_lv,f_re,f_ss,ut})

# print(ut.to_table())
# fire_dn.show()
# fire_dn.show(fontsize=15)
```

Cheating Decision Network

The following is the representation of the cheating decision shown in Figure 12.3. Someone has to decide whether to cheat at two different times. Cheating can improve grades. However, someone is watching for cheating, and if caught, results in punishment. The utility is a combination of final grade and the punishment. The decision maker finds out whether they were caught the first time when they have to decide whether to cheat the second time.

decnNetworks.py — (continued)

```
grades = ['A','B','C','F']
Watched = Variable("Watched", boolean, position=(0,0.9))
Caught1 = Variable("Caught1", boolean, position=(0.2,0.7))
Caught2 = Variable("Caught2", boolean, position=(0.6,0.7))
Punish = Variable("Punish", ["None","Suspension","Recorded"],
                  position=(0.8,0.9))
Grade_1 = Variable("Grade_1", grades, position=(0.2,0.3))
Grade_2 = Variable("Grade_2", grades, position=(0.6,0.3))
Fin_Grd = Variable("Fin_Grd", grades, position=(0.8,0.1))
Cheat_1 = DecisionVariable("Cheat_1", boolean, set(), position=(0,0.5))
Cheat_2 = DecisionVariable("Cheat_2", boolean, {Cheat_1,Caught1},
                           position=(0.4,0.5))

p_wa = Prob(Watched,[],[0.7, 0.3])
```

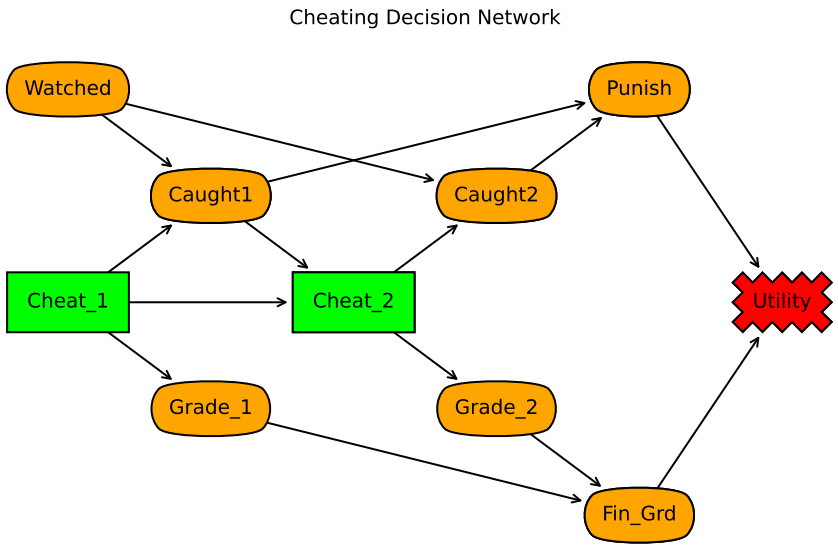


Figure 12.3: Cheating Decision Network (cheating_dn.show())

```
p_cc1 = Prob(Caught1,[Watched,Cheat_1],[[[1.0, 0.0], [0.9, 0.1]],
                                        [[1.0, 0.0], [0.5, 0.5]]])
p_cc2 = Prob(Caught2,[Watched,Cheat_2],[[[1.0, 0.0], [0.9, 0.1]],
                                        [[1.0, 0.0], [0.5, 0.5]]])
p_pun = Prob(Punish,[Caught1,Caught2],
                [[{"None":0,"Suspension":0,"Recorded":0},
                  {"None":0.5,"Suspension":0.4,"Recorded":0.1}],
                 [{"None":0.6,"Suspension":0.2,"Recorded":0.2},
                  {"None":0.2,"Suspension":0.3,"Recorded":0.3}]])
p_gr1 = Prob(Grade_1,[Cheat_1], [{'A':0.2, 'B':0.3, 'C':0.3, 'F': 0.2},
                                 {'A':0.5, 'B':0.3, 'C':0.2, 'F':0.0}])
p_gr2 = Prob(Grade_2,[Cheat_2], [{'A':0.2, 'B':0.3, 'C':0.3, 'F': 0.2},
                                 {'A':0.5, 'B':0.3, 'C':0.2, 'F':0.0}])
p_fg = Prob(Fin_Grd,[Grade_1,Grade_2],
       {'A':{'A':{'A':1.0, 'B':0.0, 'C': 0.0, 'F':0.0},
             'B': {'A':0.5, 'B':0.5, 'C': 0.0, 'F':0.0},
             'C':{'A':0.25, 'B':0.5, 'C': 0.25, 'F':0.0},
             'F':{'A':0.25, 'B':0.25, 'C': 0.25, 'F':0.25}},
        'B':{'A':{'A':0.5, 'B':0.5, 'C': 0.0, 'F':0.0},
             'B': {'A':0.0, 'B':1, 'C': 0.0, 'F':0.0},
             'C':{'A':0.0, 'B':0.5, 'C': 0.5, 'F':0.0},
             'F':{'A':0.0, 'B':0.25, 'C': 0.5, 'F':0.25}},
        'C':{'A':{'A':0.25, 'B':0.5, 'C': 0.25, 'F':0.0},
             'B': {'A':0.0, 'B':0.5, 'C': 0.5, 'F':0.0},
```

```
            'C':{'A':0.0, 'B':0.0, 'C': 1, 'F':0.0},
            'F':{'A':0.0, 'B':0.0, 'C': 0.5, 'F':0.5}},
        'F':{'A':{'A':0.25, 'B':0.25, 'C': 0.25, 'F':0.25},
            'B': {'A':0.0, 'B':0.25, 'C': 0.5, 'F':0.25},
            'C':{'A':0.0, 'B':0.0, 'C': 0.5, 'F':0.5},
            'F':{'A':0.0, 'B':0.0, 'C': 0, 'F':1.0}}})

utc = UtilityTable([Punish,Fin_Grd],
                   {'None':{'A':100, 'B':90, 'C': 70, 'F':50},
                    'Suspension':{'A':40, 'B':20, 'C': 10, 'F':0},
                    'Recorded':{'A':70, 'B':60, 'C': 40, 'F':20}},
                   position=(1,0.5))

cheating_dn = DecisionNetwork("Cheating Decision Network",
              {Punish,Caught2,Watched,Fin_Grd,Grade_2,Grade_1,Cheat_2,Caught1,Cheat_1},
              {p_wa, p_cc1, p_cc2, p_pun, p_gr1, p_gr2,p_fg,utc})

# cheating_dn.show()
# cheating_dn.show(fontsize=15)
```

Chain of 3 decisions

The following decision network represents a finite-stage fully-observable Markov decision process with a single reward (utility) at the end. It is interesting because the parents do not include all the predecessors. The methods we use will work without change on this, even though the agent does not condition on all of its previous observations and actions. The output of `ch3.show()` is shown in Figure 12.4.

decnNetworks.py — (continued)

```
S0 = Variable('S0', boolean, position=(0,0.5))
D0 = DecisionVariable('D0', boolean, {S0}, position=(1/7,0.1))
S1 = Variable('S1', boolean, position=(2/7,0.5))
D1 = DecisionVariable('D1', boolean, {S1}, position=(3/7,0.1))
S2 = Variable('S2', boolean, position=(4/7,0.5))
D2 = DecisionVariable('D2', boolean, {S2}, position=(5/7,0.1))
S3 = Variable('S3', boolean, position=(6/7,0.5))

p_s0 = Prob(S0, [], [0.5,0.5])
tr = [[[0.1, 0.9], [0.9, 0.1]], [[0.2, 0.8], [0.8, 0.2]]] # 0 is flip, 1
    is keep value
p_s1 = Prob(S1, [D0,S0], tr)
p_s2 = Prob(S2, [D1,S1], tr)
p_s3 = Prob(S3, [D2,S2], tr)

ch3U = UtilityTable([S3],[0,1], position=(7/7,0.9))

ch3 = DecisionNetwork("3-chain",
    {S0,D0,S1,D1,S2,D2,S3},{p_s0,p_s1,p_s2,p_s3,ch3U})
```

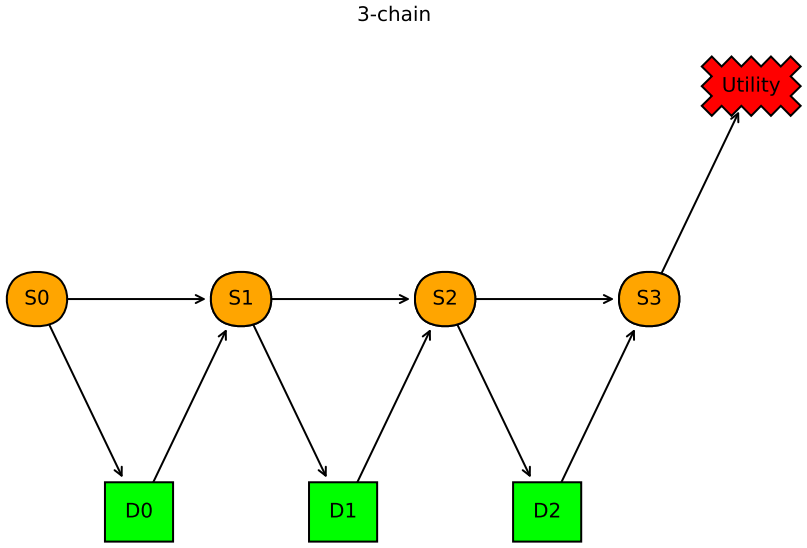


Figure 12.4: A decision network that is a chain of 3 decisions (`ch3.show()`)

```
244
245 # ch3.show()
246 # ch3.show(fontsize=15)
```

12.1.2 Decision Functions

The output of an optimization function is an optimal policy and its expected value. A policy is a list of decision functions. A decision function is the action for each decision variable as a function of its parents.

Let's represent the factor for a decision function as a dictionary.

decnNetworks.py — (continued)

```
248 class DictFactor(Factor):
249     """A factor that represents its values using a dictionary"""
250     def __init__(self, *pargs, **kwargs):
251         self.values = {}
252         Factor.__init__(self, *pargs, **kwargs)
253
254     def assign(self, assignment, value):
255         self.values[frozenset(assignment.items())] = value
256
257     def get_value(self, assignment):
258         ass = frozenset(assignment.items())
```

```
259         assert ass in self.values, f"assignment {assignment} cannot be
                evaluated"
260         return self.values[ass]
261
262 class DecisionFunction(DictFactor):
263     def __init__(self, decision, parents):
264         """ A decision function
265         decision is a decision variable
266         parents is a set of variables
267         """
268         self.decision = decision
269         self.parent = parents
270         DictFactor.__init__(self, parents, name=decision.name)
```

12.1.3 Recursive Conditioning for Decision Networks

An instance of a `RC_DN` object takes in a decision network. The query method uses recursive conditioning to compute the expected utility of the optimal policy. When it is finished, `self.opt_policy` is the optimal policy.

decnNetworks.py — (continued)

```
272 import math
273 from display import Displayable
274 from probGraphicalModels import GraphicalModel
275 from probFactors import Factor
276 from probRC import connected_components
277
278 class RC_DN(Displayable):
279     """The class that finds the optimal policy for a decision network.
280
281     dn is graphical model to query
282     """
283
284     def __init__(self, dn):
285         self.dn = dn
286         self.cache = {(frozenset(), frozenset()):1}
287         ## self.max_display_level = 3
288
289     def optimize(self, split_order=None, algorithm=None):
290         """computes expected utility, and creates optimal decision
                functions, where
291         elim_order is a list of the non-observed non-query variables in dn
292         algorithm is the (search algorithm to use). Default is self.rc
293         """
294         if algorithm is None:
295             algorithm = self.rc
296         if split_order == None:
297             split_order = self.dn.split_order()
298         self.opt_policy = {v:DecisionFunction(v, v.parents)
299                            for v in self.dn.variables
```

```
                            if isinstance(v,DecisionVariable)}
        return algorithm({}, self.dn.factors, split_order)

    def show_policy(self):
        print('\n'.join(df.to_table() for df in self.opt_policy.values()))
```

The following is the simplest search-based algorithm. It is exponential in the number of variables, so is not very useful. However, it is simple, and helpful to understand before looking at the more complicated algorithm. Note that the above code does not call `rc0`; you will need to change the `self.rc` to `self.rc0` in above code to use it.

decnNetworks.py — (continued)

```
    def rc0(self, context, factors, split_order):
        """simplest search algorithm
        context is a variable:value dictionary
        factors is a set of factors
        split_order is a list of variables in factors that are not in
            context
        """
        self.display(3,"calling rc0,",(context,factors),"with
            SO",split_order)
        if not factors:
            return 1
        elif to_eval := {fac for fac in factors if
            fac.can_evaluate(context)}:
            self.display(3,"rc0 evaluating factors",to_eval)
            val = math.prod(fac.get_value(context) for fac in to_eval)
            return val * self.rc0(context, factors-to_eval, split_order)
        else:
            var = split_order[0]
            self.display(3, "rc0 branching on", var)
            if isinstance(var,DecisionVariable):
                assert set(context) <= set(var.parents), f"cannot optimize
                    {var} in context {context}"
                maxres = -math.inf
                for val in var.domain:
                    self.display(3,"In rc0, branching on",var,"=",val)
                    newres = self.rc0({var:val}|context, factors,
                        split_order[1:])
                    if newres > maxres:
                        maxres = newres
                        theval = val
                self.opt_policy[var].assign(context,theval)
                return maxres
            else:
                total = 0
                for val in var.domain:
                    total += self.rc0({var:val}|context, factors,
                        split_order[1:])
                self.display(3, "rc0 branching on", var,"returning", total)
```

```
338             return total
```

We can combine the optimization for decision networks above, with the improvements of recursive conditioning used for graphical models (Section 9.7, page 222).

decnNetworks.py — (continued)

```
340     def rc(self, context, factors, split_order):
341         """ returns the number sum_{split_order} prod_{factors} given
                assignments in context
342         context is a variable:value dictionary
343         factors is a set of factors
344         split_order is a list of variables in factors that are not in
                context
345         """
346         self.display(3,"calling rc,",(context,factors))
347         ce = (frozenset(context.items()), frozenset(factors)) # key for the
                cache entry
348         if ce in self.cache:
349             self.display(2,"rc cache lookup",(context,factors))
350             return self.cache[ce]
351 #       if not factors: # no factors; needed if you don't have forgetting
        and caching
352 #           return 1
353         elif vars_not_in_factors := {var for var in context
354                                      if not any(var in fac.variables for
                                           fac in factors)}:
355             # forget variables not in any factor
356             self.display(3,"rc forgetting variables", vars_not_in_factors)
357             return self.rc({key:val for (key,val) in context.items()
358                             if key not in vars_not_in_factors},
359                            factors, split_order)
360         elif to_eval := {fac for fac in factors if
                fac.can_evaluate(context)}:
361             # evaluate factors when all variables are assigned
362             self.display(3,"rc evaluating factors",to_eval)
363             val = math.prod(fac.get_value(context) for fac in to_eval)
364             if val == 0:
365                 return 0
366             else:
367                 return val * self.rc(context, {fac for fac in factors if fac
                        not in to_eval}, split_order)
368         elif len(comp := connected_components(context, factors,
                split_order)) > 1:
369             # there are disconnected components
370             self.display(2,"splitting into connected components",comp)
371             return(math.prod(self.rc(context,f,eo) for (f,eo) in comp))
372         else:
373             assert split_order, f"split_order empty rc({context},{factors})"
374             var = split_order[0]
375             self.display(3, "rc branching on", var)
```

```
            if isinstance(var,DecisionVariable):
                assert set(context) <= set(var.parents), f"cannot optimize
                    {var} in context {context}"
                maxres = -math.inf
                for val in var.domain:
                    self.display(3,"In rc, branching on",var,"=",val)
                    newres = self.rc({var:val}|context, factors,
                        split_order[1:])
                    if newres > maxres:
                        maxres = newres
                        theval = val
                self.opt_policy[var].assign(context,theval)
                self.cache[ce] = maxres
                return maxres
            else:
                total = 0
                for val in var.domain:
                    total += self.rc({var:val}|context, factors,
                        split_order[1:])
                self.display(3, "rc branching on", var,"returning", total)
                self.cache[ce] = total
                return total
```

Here is how to run the optimizer on the example decision networks:

decnNetworks.py — (continued)

```
# Umbrella decision network
#urc = RC_DN(umbrella_dn)
#urc.optimize(algorithm=urc.rc0) #RC0
#urc.optimize()  #RC
#urc.show_policy()

#rc_fire = RC_DN(fire_dn)
#rc_fire.optimize()
#rc_fire.show_policy()

#rc_cheat = RC_DN(cheating_dn)
#rc_cheat.optimize()
#rc_cheat.show_policy()

#rc_ch3 = RC_DN(ch3)
#rc_ch3.optimize()
#rc_ch3.show_policy()
# rc_ch3.optimize(algorithm=rc_ch3.rc0) # why does that happen?
```

12.1.4 Variable elimination for decision networks

`VE_DN` is variable elimination for decision networks. The method *optimize* is used to optimize all the decisions. Note that *optimize* requires a legal elimination ordering of the random and decision variables, otherwise it will give an

exception. (A decision node can only be maximized if the variables that are not its parents have already been eliminated.)

decnNetworks.py — (continued)

```
from probVE import VE

class VE_DN(VE):
    """Variable Elimination for Decision Networks"""
    def __init__(self,dn=None):
        """dn is a decision network"""
        VE.__init__(self,dn)
        self.dn = dn

    def optimize(self,elim_order=None,obs={}):
        if elim_order == None:
               elim_order = reversed(self.dn.split_order())
        self.opt_policy = {}
        proj_factors = [self.project_observations(fact,obs)
                        for fact in self.dn.factors]
        for v in elim_order:
            if isinstance(v,DecisionVariable):
                to_max = [fac for fac in proj_factors
                         if v in fac.variables and set(fac.variables) <=
                             v.all_vars]
                assert len(to_max)==1, "illegal variable order
                    "+str(elim_order)+" at "+str(v)
                newFac = FactorMax(v, to_max[0])
                self.opt_policy[v]=newFac.decision_fun
                proj_factors = [fac for fac in proj_factors if fac is not
                    to_max[0]]+[newFac]
                self.display(2,"maximizing",v )
                self.display(3,newFac)
            else:
                proj_factors = self.eliminate_var(proj_factors, v)
        assert len(proj_factors)==1,"Should there be only one element of
            proj_factors?"
        return proj_factors[0].get_value({})

    def show_policy(self):
        print('\n'.join(df.to_table() for df in self.opt_policy.values()))
```

decnNetworks.py — (continued)

```
class FactorMax(TabFactor):
    """A factor obtained by maximizing a variable in a factor.
    Also builds a decision_function. This is based on FactorSum.
    """

    def __init__(self, dvar, factor):
        """dvar is a decision variable.
        factor is a factor that contains dvar and only parents of dvar
```

```
        """
        self.dvar = dvar
        self.factor = factor
        vars = [v for v in factor.variables if v is not dvar]
        Factor.__init__(self,vars)
        self.values = {}
        self.decision_fun = DecisionFunction(dvar, dvar.parents)

    def get_value(self,assignment):
        """lazy implementation: if saved, return saved value, else compute
            it"""
        new_asst = {x:v for (x,v) in assignment.items() if x in
            self.variables}
        asst = frozenset(new_asst.items())
        if asst in self.values:
            return self.values[asst]
        else:
            max_val = float("-inf") # -infinity
            for elt in self.dvar.domain:
                fac_val = self.factor.get_value(assignment|{self.dvar:elt})
                if fac_val>max_val:
                    max_val = fac_val
                    best_elt = elt
            self.values[asst] = max_val
            self.decision_fun.assign(assignment, best_elt)
            return max_val
```

Here are some example queries:

decnNetworks.py — (continued)

```
# Example queries:
# vf = VE_DN(fire_dn)
# vf.optimize()
# vf.show_policy()

# VE_DN.max_display_level = 3 # if you want to show lots of detail
# vc = VE_DN(cheating_dn)
# vc.optimize()
# vc.show_policy()

def test(dn):
    rc0dn = RC_DN(dn)
    rc0v = rc0dn.optimize(algorithm=rc0dn.rc0)
    rcdn = RC_DN(dn)
    rcv = rcdn.optimize()
    assert abs(rc0v-rcv)<1e-10, f"rc0 produces {rc0v}; rc produces {rcv}"
    vedn = VE_DN(dn)
    vev = vedn.optimize()
    assert abs(vev-rcv)<1e-10, f"VE_DN produces {vev}; RC produces {rcv}"
    print(f"passed unit test. rc0, rc and VE gave same result for {dn}")

```

```
if __name__ == "__main__":
    test(fire_dn)
```

12.2 Markov Decision Processes

The following represent a **Markov decision process** (**MDP**) directly, rather than using the recursive conditioning or variable elimination code.

mdpProblem.py — Representations for Markov Decision Processes

```
import random
from display import Displayable
from utilities import argmaxd

class MDP(Displayable):
    """A Markov Decision Process. Must define:
    title a string that gives the title of the MDP
    states the set (or list) of states
    actions the set (or list) of actions
    discount a real-valued discount
    """

    def __init__(self, title, states, actions, discount, init=0):
        self.title = title
        self.states = states
        self.actions = actions
        self.discount = discount
        self.initv = self.V = {s:init for s in self.states}
        self.initq = self.Q = {s: {a: init for a in self.actions} for s in
            self.states}

    def P(self,s,a):
        """Transition probability function
        returns a dictionary of {s1:p1} such that P(s1 | s,a)=p1,
                and other probabilities are zero.
        """
        raise NotImplementedError("P") # abstract method

    def R(self,s,a):
        """Reward function R(s,a)
        returns the expected reward for doing a in state s.
        """
        raise NotImplementedError("R") # abstract method
```

Two state partying example (Example 12.29 in Poole and Mackworth [2023]):

mdpExamples.py — MDP Examples

```
from mdpProblem import MDP, ProblemDomain, distribution
from mdpGUI import GridDomain
```

```
13 import matplotlib.pyplot as plt
14
15 class partyMDP(MDP):
16     """Simple 2-state, 2-Action Partying MDP Example"""
17     def __init__(self, discount=0.9):
18         states = {'healthy','sick'}
19         actions = {'relax', 'party'}
20         MDP.__init__(self, "party MDP", states, actions, discount)
21
22     def R(self,s,a):
23         "R(s,a)"
24         return { 'healthy': {'relax': 7, 'party': 10},
25                  'sick':   {'relax': 0, 'party': 2 }}[s][a]
26
27     def P(self,s,a):
28         "returns a dictionary of {s1:p1} such that P(s1 | s,a)=p1. Other
               probabilities are zero."
29         phealthy = { # P('healthy' | s, a)
30                     'healthy': {'relax': 0.95, 'party': 0.7},
31                     'sick': {'relax': 0.5, 'party': 0.1 }}[s][a]
32         return {'healthy':phealthy, 'sick':1-phealthy}
```

The `distribution` class is used to represent distributions as they are being created. Probability distributions are represented as *item* : *value* dictionaries. When being constructed, adding an *item* : *value* to the dictionary has to act differently when the item is already in the dictionary and when it isn't. The `add_prob` method works whether the item is in the dictionary or not.

mdpProblem.py — (continued)

```
44 class distribution(dict):
45     """A distribution is an item:prob dictionary.
46     Probabilities are added using add_prop.
47     """
48     def __init__(self,d):
49         dict.__init__(self,d)
50
51     def add_prob(self, item, pr):
52         """adds a probability to a distribution.
53         Like dictionary assignment, but if item is already there, the
               values are summed
54         """
55         if item in self:
56             self[item] += pr
57         else:
58             self[item] = pr
59         return self
```

12.2.1 Problem Domains

An MDP does not contain enough information to simulate a domain, because

(a) the rewards and resulting state can be correlated (e.g., in the grid domains below, crashing into a wall results in both a negative reward and the agent not moving), and

(b) it represents the *expected* reward (e.g., a reward of 1 is has the same expected value as a reward of 100 with probability 1/100 and 0 otherwise, but these are different in a simulation).

A problem domain represents a problem as a function `result` from states and actions into a distribution of $(state, reward)$ pairs. This can be a subclass of `MDP` because it implements `R` and `P`. A problem domain also specifies an initial state and coordinate information used by the graphical user interfaces.

mdpProblem.py — (continued)

```
class ProblemDomain(MDP):
    """A ProblemDomain implements
    self.result(state, action) -> {(reward, state):probability}.
    Other pairs have probability are zero.
    The probabilities must sum to 1.
    """
    def __init__(self, title, states, actions, discount,
                 initial_state=None, x_dim=0, y_dim = 0,
                 vinit=0, offsets={}):
        """A problem domain
        * title is list of titles
        * states is the list of states
        * actions is the list of actions
        * discount is the discount factor
        * initial_state is the state the agent starts at (for simulation)
            if known
        * x_dim and y_dim are the dimensions used by the GUI to show the
            states in 2-dimensions
        * vinit is the initial value
        * offsets is a {action:(x,y)} map which specifies how actions are
            displayed in GUI
        """
        MDP.__init__(self, title, states, actions, discount)
        if initial_state is not None:
            self.state = initial_state
        else:
            self.state = random.choice(states)
        self.vinit = vinit # value to reset v,q to
        # The following are for the GUI:
        self.x_dim = x_dim
        self.y_dim = y_dim
        self.offsets = offsets

    def state2pos(self,state):
        """When displaying as a grid, this specifies how the state is
            mapped to (x,y) position.
        The default is for domains where the (x,y) position is the state
```

```
        """
        return state

    def state2goal(self,state):
        """When displaying as a grid, this specifies how the state is
            mapped to goal position.
        The default is for domains where there is no goal
        """
        return None

    def pos2state(self,pos):
        """When displaying as a grid, this specifies how the state is
            mapped to (x,y) position.
        The default is for domains where the (x,y) position is the state
        """
        return pos

    def P(self, state, action):
        """Transition probability function
        returns a dictionary of {s1:p1} such that P(s1 | state,action)=p1.
        Other probabilities are zero.
        """
        res = self.result(state, action)
        acc = 1e-6 # accuracy for test of equality
        assert 1-acc<sum(res.values())<1+acc, f"result({state},{action})
            not a distribution, sum={sum(res.values())}"
        dist = distribution({})
        for ((r,s),p) in res.items():
            dist.add_prob(s,p)
        return dist

    def R(self, state, action):
        """Reward function R(s,a)
        returns the expected reward for doing a in state s.
        """
        return sum(r*p for ((r,s),p) in self.result(state, action).items())
```

Tiny Game

The next example is the tiny game from Example 13.1 and Figure 13.1 of Poole and Mackworth [2023], shown here as Figure 12.5. There are 6 states and 4 actions. The state is represented as (x, y) where x counts from zero from the left, and y counts from zero upwards, so the state $(0, 0)$ is on the bottom-left. The actions are `upC` for up-careful, `upR` for up-risky, `left`, and `right`. Going left from $(0, 2)$ results in a reward of 10 and ending up in state $(0, 0)$; going left from $(0, 1)$ results in a reward of -100 and staying there. Up-risky goes up but with a chance of going left or right. Up careful goes up, but has a reward of -1. Left and right are deterministic. Crashing into a wall results in a reward of -1 and staying still.

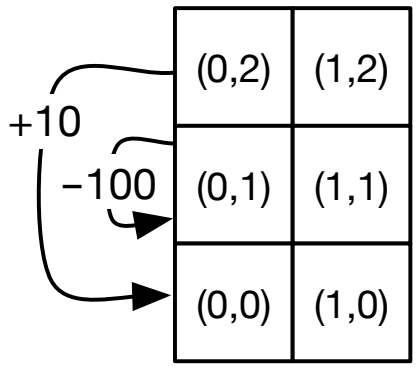


Figure 12.5: Tiny game

(Note that `GridDomain` means that it can be shown with the MDP GUI in Section 12.2.3).

mdpExamples.py — (continued)

```
34 class MDPtiny(ProblemDomain, GridDomain):
35     def __init__(self, discount=0.9):
36         x_dim = 2  # x-dimension
37         y_dim = 3
38         ProblemDomain.__init__(self,
39             "Tiny MDP", # title
40             [(x,y) for x in range(x_dim) for y in range(y_dim)], #states
41             ['right', 'upC', 'left', 'upR'], #actions
42             discount,
43             x_dim=x_dim, y_dim = y_dim,
44             offsets = {'right':(0.25,0), 'upC':(0,-0.25), 'left':(-0.25,0),
                   'upR':(0,0.25)}
45             )
46
47     def result(self, state, action):
48         """return a dictionary of {(r,s):p} where p is the probability of
                reward r, state s
49         a state is an (x,y) pair
50         """
51         (x,y) = state
52         right = (-x,(1,y)) # reward is -1 if x was 1
53         left = (0,(0,y)) if x==1 else [(-1,(0,0)), (-100,(0,1)),
               (10,(0,0))][y]
54         up = (0,(x,y+1)) if y<2 else (-1,(x,y))
55         if action == 'right':
56             return {right:1}
57         elif action == 'upC':
58             (r,s) = up
```

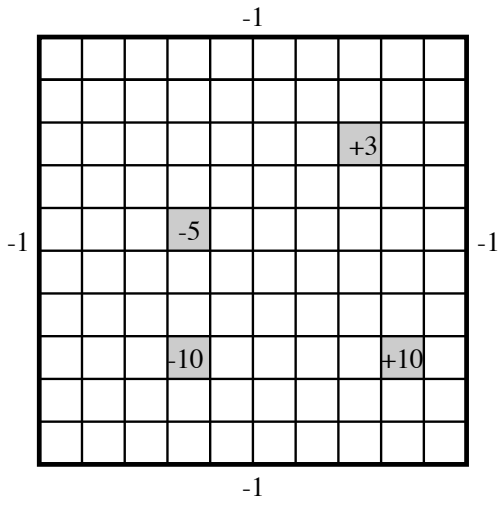


Figure 12.6: Grid world

```
59            return {(r-1,s):1}
60        elif action == 'left':
61           return {left:1}
62        elif action == 'upR':
63            return distribution({left:
                  0.1}).add_prob(right,0.1).add_prob(up,0.8)
64            # Exercise: what is wrong with return {left: 0.1, right:0.1,
                  up:0.8}
65
66 # To show GUI do
67 #  MDPtiny().viGUI()
```

Grid World

Here is the domain of Example 12.30 of Poole and Mackworth [2023], shown here in Figure 12.6. A state is represented as (x, y) where x counts from zero from the left, and y counts from zero upwards, so the state $(0, 0)$ is on the bottom-left.

mdpExamples.py — (continued)

```
69 class grid(ProblemDomain, GridDomain):
70     """ x_dim * y_dim grid with rewarding states"""
71     def __init__(self, discount=0.9, x_dim=10, y_dim=10):
72         ProblemDomain.__init__(self,
73             "Grid World",
74             [(x,y) for x in range(y_dim) for y in range(y_dim)], #states
75             ['up', 'down', 'right', 'left'], #actions
76             discount,
77             x_dim = x_dim, y_dim = y_dim,
```

```
78          offsets = {'right':(0.25,0), 'up':(0,0.25), 'left':(-0.25,0),
                'down':(0,-0.25)})
79      self.rewarding_states = {(3,2):-10, (3,5):-5, (8,2):10, (7,7):3 }
80      self.fling_states = {(8,2), (7,7)} # assumed a subset of
            rewarding_states
81
82  def intended_next(self,s,a):
83      """returns the (reward, state) in the direction a.
84      This is where the agent will end up if to goes in its
            intended_direction
85          (which it does with probability 0.7).
86      """
87      (x,y) = s
88      if a=='up':
89          return (0, (x,y+1)) if y+1 < self.y_dim else (-1, (x,y))
90      if a=='down':
91          return (0, (x,y-1)) if y > 0 else (-1, (x,y))
92      if a=='right':
93          return (0, (x+1,y)) if x+1 < self.x_dim else (-1, (x,y))
94      if a=='left':
95          return (0, (x-1,y)) if x > 0 else (-1, (x,y))
96
97  def result(self,s,a):
98      """return a dictionary of {(r,s):p} where p is the probability of
            reward r, state s.
99      a state is an (x,y) pair
100     """
101     r0 = self.rewarding_states[s] if s in self.rewarding_states else 0
102     if s in self.fling_states:
103         return {(r0,(0,0)): 0.25, (r0,(self.x_dim-1,0)):0.25,
104                     (r0,(0,self.y_dim-1)):0.25,
                        (r0,(self.x_dim-1,self.y_dim-1)):0.25}
105     dist = distribution({})
106     for a1 in self.actions:
107         (r1,s1) = self.intended_next(s,a1)
108         rs = (r1+r0, s1)
109         p = 0.7 if a1==a else 0.1
110         dist.add_prob(rs,p)
111     return dist
```

Figure 12.7 shows the immediate expected reward for each of the 100 states. This was generated using `grid().viGUI()` and carrying out one step.

Monster Game

This is for the game depicted in Figure 12.8 (Example 13.2 of Poole and Mackworth [2023]). There are 25 locations where the agent can be, there can be no prize or there can be a prize in one of the corners ($P_1 \ldots P_4$). The agent only gets a positive reward when gets to the prize. The agent can be damaged or undamaged. There are possible monsters at the locations marked with M. If

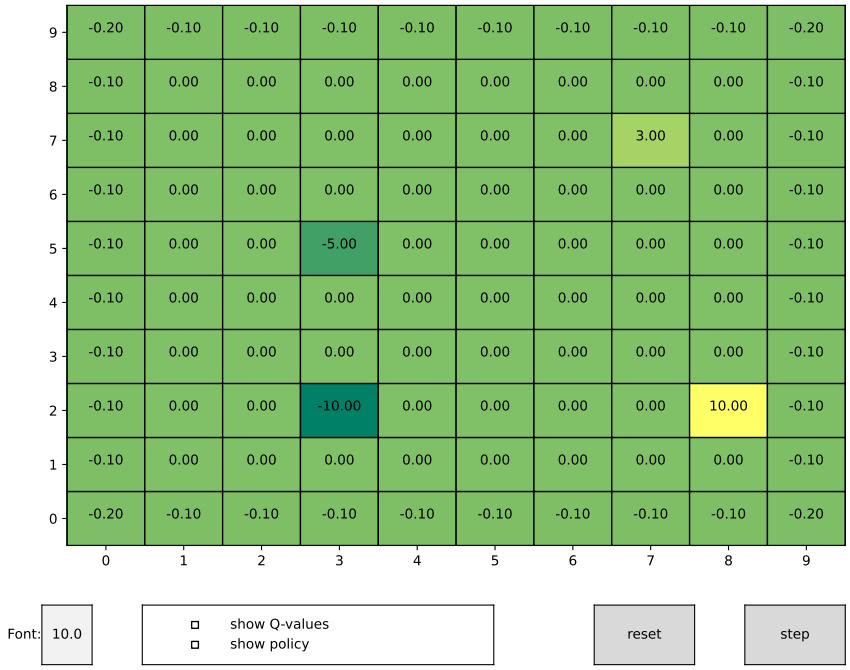


Figure 12.7: Grid world GUI: `grid().viGUI()`

the agent lands on a monster when it is undamaged, it gets damaged. If the agent lands on a monster when it is damaged, it gets a negative reward. It can get undamaged by going to the location marked with R. It gets a negative reward by crashing into a wall. There are $25 * 5 * 2 = 250$ states. There are 4 actions, *up*, *down*, *left*, and *right*; the agent generally goes in the direction of the action, but has a chance of going in one of the other directions.

mdpExamples.py — (continued)

```
113 class Monster_game(ProblemDomain, GridDomain):
114
115     vwalls = [(0,3), (0,4), (1,4)] # vertical walls right of these locations
116     crash_reward = -1
117
118     prize_locs = [(0,0), (0,4), (4,0), (4,4)]
119     prize_apears_prob = 0.3
120     prize_reward = 10
121
122     monster_locs = [(0,1), (1,1), (2,3), (3,1), (4,2)]
123     monster_appears_prob = 0.4
124     monster_reward_when_damaged = -10
125     repair_stations = [(1,4)]
```

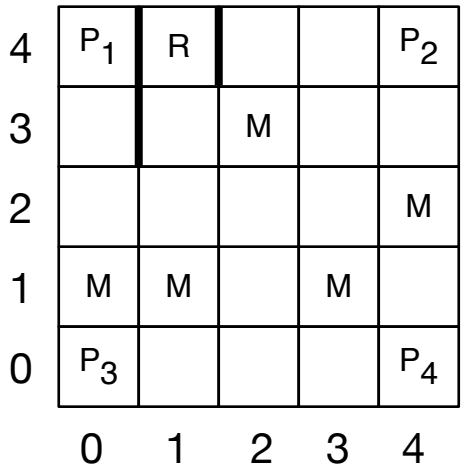


Figure 12.8: Monster game

```
126
127     def __init__(self, discount=0.9):
128         x_dim = 5
129         y_dim = 5
130             # which damaged and prize to show
131         ProblemDomain.__init__(self,
132             "Monster Game",
133             [(x,y,damaged,prize)
134                  for x in range(x_dim)
135                  for y in range(y_dim)
136                  for damaged in [False,True]
137                  for prize in [None]+self.prize_locs], #states
138             ['up', 'down', 'right', 'left'], #actions
139             discount,
140             x_dim = x_dim, y_dim = y_dim,
141             offsets = {'right':(0.25,0), 'up':(0,0.25), 'left':(-0.25,0),
                    'down':(0,-0.25)})
142         self.state = (2,2,False,None)
143
144     def intended_next(self,xy,a):
145         """returns the (reward, (x,y)) in the direction a.
146         This is where the agent will end up if to goes in its
                intended_direction
147              (which it does with probability 0.7).
148         """
149         (x,y) = xy # original x-y position
150         if a=='up':
151             return (0, (x,y+1)) if y+1 < self.y_dim else
                    (self.crash_reward, (x,y))
152         if a=='down':
153             return (0, (x,y-1)) if y > 0 else (self.crash_reward, (x,y))
```

```
        if a=='right':
            if (x,y) in self.vwalls or x+1==self.x_dim: # hit wall
                return (self.crash_reward, (x,y))
            else:
                return (0, (x+1,y))
        if a=='left':
            if (x-1,y) in self.vwalls or x==0: # hit wall
                        return (self.crash_reward, (x,y))
            else:
                return (0, (x-1,y))

    def result(self,s,a):
        """return a dictionary of {(r,s):p} where p is the probability of
            reward r, state s.
        a state is an (x,y) pair
        """
        (x,y,damaged,prize) = s
        dist = distribution({})
        for a1 in self.actions: # possible results
            mp = 0.7 if a1==a else 0.1
            mr,(xn,yn) = self.intended_next((x,y),a1)
            if (xn,yn) in self.monster_locs:
                if damaged:
                    dist.add_prob((mr+self.monster_reward_when_damaged,(xn,yn,True,prize)),
                        mp*self.monster_appears_prob)
                    dist.add_prob((mr,(xn,yn,True,prize)),
                        mp*(1-self.monster_appears_prob))
                else:
                   dist.add_prob((mr,(xn,yn,True,prize)),
                       mp*self.monster_appears_prob)
                   dist.add_prob((mr,(xn,yn,False,prize)),
                       mp*(1-self.monster_appears_prob))
            elif (xn,yn) == prize:
                dist.add_prob((mr+self.prize_reward,(xn,yn,damaged,None)),
                    mp)
            elif (xn,yn) in self.repair_stations:
                dist.add_prob((mr,(xn,yn,False,prize)), mp)
            else:
                dist.add_prob((mr,(xn,yn,damaged,prize)), mp)
        if prize is None:
            res = distribution({})
            for (r,(x2,y2,d,p2)),p in dist.items():
                res.add_prob((r,(x2,y2,d,None)),
                    p*(1-self.prize_apears_prob))
                for pz in self.prize_locs:
                    res.add_prob((r,(x2,y2,d,pz)),
                        p*self.prize_apears_prob/len(self.prize_locs))
            return res
        else:
            return dist
```

```
196
197     def state2pos(self, state):
198         """When displaying as a grid, this specifies how the state is
                mapped to (x,y) position.
199         The default is for domains where the (x,y) position is the state
200         """
201         (x,y,d,p) = state
202         return (x,y)
203
204     def pos2state(self, pos):
205         """When displaying as a grid, this specifies how the state is
                mapped to (x,y) position.
206         """
207         (x,y) = pos
208         (xs, ys, damaged, prize) = self.state
209         return (x, y, damaged, prize)
210
211     def state2goal(self,state):
212         """the (x,y) position for the goal
213         """
214         (x, y, damaged, prize) = state
215         return prize
216
217 # value iteration GUI for Monster game:
218 # mg = Monster_game()
219 # mg.viGUI() # then run vi a few times
220 # to see other states, exit the GUI
221 # mg.state = (2,2,True,(4,4)) # or other damaged/prize states
222 # mg.viGUI()
```

12.2.2 Value Iteration

The following implements value iteration for Markov decision processes.

A Q function is represented as a dictionary so $Q[s][a]$ is the value for doing action a in state s. The value function is represented as a dictionary so $V[s]$ is the value of state s. Policy π is represented as a dictionary where $pi[s]$, where s is a state, returns the action.

Note that the following defines `vi` to be a method in `MDP`.

mdpProblem.py — (continued)

```
128 def vi(self, n):
129         """carries out n iterations of value iteration, updating value
                function self.V
130         Returns a Q-function, value function, policy
131         """
132         self.display(3,f"calling vi({n})")
133         for i in range(n):
134             self.Q = {s: {a: self.R(s,a)
135                           +self.discount*sum(p1*self.V[s1]
```

```
                                                for (s1,p1) in
                                                    self.P(s,a).items())
                        for a in self.actions}
                    for s in self.states}
            self.V = {s: max(self.Q[s][a] for a in self.actions)
                      for s in self.states}
        self.pi = {s: argmaxd(self.Q[s])
                   for s in self.states}
        return self.Q, self.V, self.pi

MDP.vi = vi
```

The following shows how this can be used.

mdpExamples.py — (continued)

```
## Testing value iteration
# Try the following:
# pt = partyMDP(discount=0.9)
# pt.vi(1)
# pt.vi(100)
# partyMDP(discount=0.99).vi(100)
# partyMDP(discount=0.4).vi(100)

# gr = grid(discount=0.9)
# gr.viGUI()
# q,v,pi = gr.vi(100)
# q[(7,2)]
```

12.2.3 Value Iteration GUI for Grid Domains

A `GridDomain` is a domain where the states can be mapped into (x, y) positions, and the actions can be mapped into up-down-left-right. They are special because the `viGUI()` method to interact with them. It requires the following values/methods be defined:

- `self.x_dim` and `self.y_dim` define the dimensions of the grid (so the states are `(x,y)`, where $0 \leq x <$ `self.x_dim` and $0 \leq y <$ `self.y_dim`.
- `self.state2pos(state)]` gives the `(x,y)` position of `state`. The default is that that states are already `(x,y)` positions.
- `self.state2goal(state)]` gives the `(x,y)` position of the goal in `state`. The default is `None`.
- `self.pos2state(pos)]` where `pos` is an `(x,y)` pair, gives the state that is shown at position `(x,y)`. When the state contain more information than the `(x,y)` pair, the extra information is taken from `self.state`.
- `self.offsets[a]` defines where to display action `a`, as (x, y) offset for action `a` when displaying Q-values.

mdpGUI.py — GUI for value iteration in MDPs

```
import matplotlib.pyplot as plt
from matplotlib.widgets import Button, CheckButtons, TextBox
from mdpProblem import MDP

class GridDomain(object):

    def viGUI(self):
        fig,self.ax = plt.subplots()
        plt.subplots_adjust(bottom=0.2)
        stepB = Button(fig.add_axes([0.8,0.05,0.1,0.075]), "step")
        stepB.on_clicked(self.on_step)
        resetB = Button(fig.add_axes([0.65,0.05,0.1,0.075]), "reset")
        resetB.on_clicked(self.on_reset)
        self.qcheck = CheckButtons(fig.add_axes([0.2,0.05,0.35,0.075]),
                                   ["show Q-values","show policy"])
        self.qcheck.on_clicked(self.show_vals)
        self.font_box = TextBox(fig.add_axes([0.1,0.05,0.05,0.075]),
                                "Font:", textalignment="center")
        self.font_box.on_submit(self.set_font_size)
        self.font_box.set_val(str(plt.rcParams['font.size']))
        self.show_vals(None)
        plt.show()

    def set_font_size(self, s):
        plt.rcParams.update({'font.size': eval(s)})
        plt.draw()

    def show_vals(self,event):
        self.ax.cla() # clear the axes

        array = [[self.V[self.pos2state((x,y))] for x in range(self.x_dim)]
                                            for y in range(self.y_dim)]
        self.ax.pcolormesh([x-0.5 for x in range(self.x_dim+1)],
                           [y-0.5 for y in range(self.y_dim+1)],
                           array, edgecolors='black',cmap='summer')
            # for cmap see
                https://matplotlib.org/stable/tutorials/colors/colormaps.html
        if self.qcheck.get_status()[1]: # "show policy"
               for x in range(self.x_dim):
                  for y in range(self.y_dim):
                     state = self.pos2state((x,y))
                     maxv = max(self.Q[state][a] for a in self.actions)
                     for a in self.actions:
                         if self.Q[state][a] == maxv:
                             # draw arrow in appropriate direction
                             xoff, yoff = self.offsets[a]
                             self.ax.arrow(x,y,xoff*2,yoff*2,
                                  color='red',width=0.05, head_width=0.2,
                                  length_includes_head=True)
```

```
        if self.qcheck.get_status()[0]: # "show q-values"
            self.show_q(event)
        else:
            self.show_v(event)
        self.ax.set_xticks(range(self.x_dim))
        self.ax.set_xticklabels(range(self.x_dim))
        self.ax.set_yticks(range(self.y_dim))
        self.ax.set_yticklabels(range(self.y_dim))
        plt.draw()

    def on_step(self,event):
        self.step()
        self.show_vals(event)

    def step(self):
        """The default step is one step of value iteration"""
        self.vi(1)

    def show_v(self,event):
        """show values"""
        for x in range(self.x_dim):
            for y in range(self.y_dim):
                state = self.pos2state((x,y))
                self.ax.text(x,y,"{val:.2f}".format(val=self.V[state]),ha='center')

    def show_q(self,event):
        """show q-values"""
        for x in range(self.x_dim):
            for y in range(self.y_dim):
                state = self.pos2state((x,y))
                for a in self.actions:
                    xoff, yoff = self.offsets[a]
                    self.ax.text(x+xoff,y+yoff,
                                 "{val:.2f}".format(val=self.Q[state][a]),ha='center')

    def on_reset(self,event):
        self.V = {s:self.vinit for s in self.states}
        self.Q = {s: {a: self.vinit for a in self.actions} for s in
            self.states}
        self.show_vals(event)

# to use the GUI do some of:
import mdpExamples
# mdpExamples.MDPtiny(discount=0.9).viGUI()
# mdpExamples.grid(discount=0.9).viGUI()
# mdpExamples.Monster_game(discount=0.9).viGUI() # see mdpExamples.py

if __name__ == "__main__":
    print("Try: mdpExamples.MDPtiny(discount=0.9).viGUI()")
```

Figure 12.9 shows the user interface for the tiny domain, which can be ob-

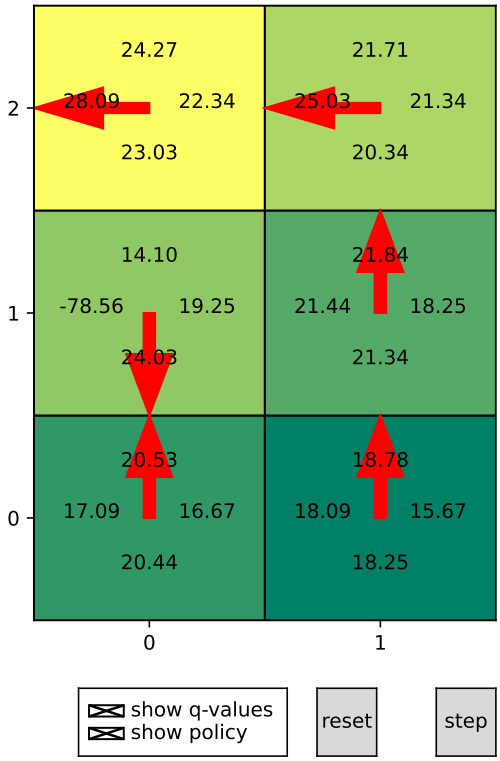


Figure 12.9: Interface for tiny example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The leftmost number is for the left action; the rightmost number is for the right action; the uppermost is for the *upR* (up-risky) action and the lowest number is for the *upC* action. The arrow points to the action(s) with the maximum Q-value. Use `MDPtiny().viGUI()` after loading `mdpExamples.py`

tained using
`MDPtiny(discount=0.9).viGUI()`
resizing it, checking "show q-values" and "show policy", and clicking "step" a few times.

To run the demo in class do:
```
% python -i mdpExamples.py
MDPtiny(discount=0.9).viGUI()
```

Figure 12.10 shows the user interface for the grid domain, which can be obtained using
`grid(discount=0.9).viGUI()`

resizing it, checking "show q-values" and "show policy", and clicking "step" a few times.

Figure 12.11 shows the optimal policy and Q-values after convergence (clicking "step" more does not change the Q-values) for the states where the agent is damaged and the goal is in the top-right. The are 10 times as many states as positions, so we can't show them all. See the commented out lines at the end of the Monster game code to reproduce this figure.

Exercise 12.1 Computing q before v may seem like a waste of space because we don't need to store q in order to compute the value function or the policy. Change the algorithm so that it loops through the states and actions once per iteration, and only stores the value function and the policy. Note that to get the same results as before, you would need to make sure that you use the previous value of v in the computation not the current value of v. Does using the current value of v hurt the algorithm or make it better (in approaching the actual value function)?

12.2.4 Asynchronous Value Iteration

This implements asynchronous value iteration, storing Q.

A Q function is represented using $Q[s][a]$ as the value for doing action with a in state s.

mdpProblem.py — (continued)

```
147 def avi(self,n):
148         states = list(self.states)
149         actions = list(self.actions)
150         for i in range(n):
151             s = random.choice(states)
152             a = random.choice(actions)
153             self.Q[s][a] = (self.R(s,a) + self.discount *
154                             sum(p1 * max(self.Q[s1][a1]
155                                          for a1 in self.actions)
156                                 for (s1,p1) in self.P(s,a).items()))
157         return self.Q
158 
159 # make this a method for the MPD class:
160 MDP.avi = avi
```

The following shows how `avi` can be used.

mdpExamples.py — (continued)

```
238 ## Testing asynchronous value iteration
239 # Try the following:
240 # pt = partyMDP(discount=0.9)
241 # pt.avi(10)
242 # pt.vi(1000)
243 
244 # gr = grid(discount=0.9)
245 # q = gr.avi(100000)
246 # q[(7,2)]
```

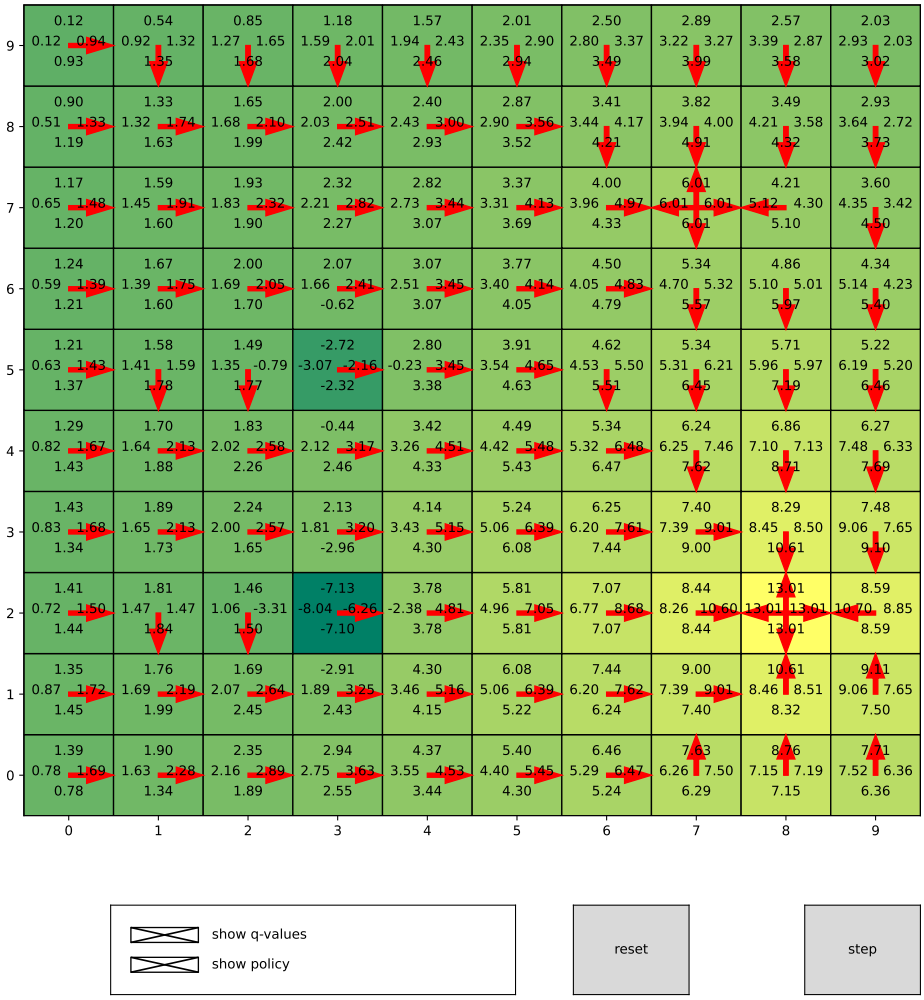


Figure 12.10: Interface for grid example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The leftmost number is for the left action; the rightmost number is for the right action; the uppermost is for the up action and the lowest number is for the down action. The arrow points to the action(s) with the maximum Q-value. From `grid(discount=0.9).viGUI()`

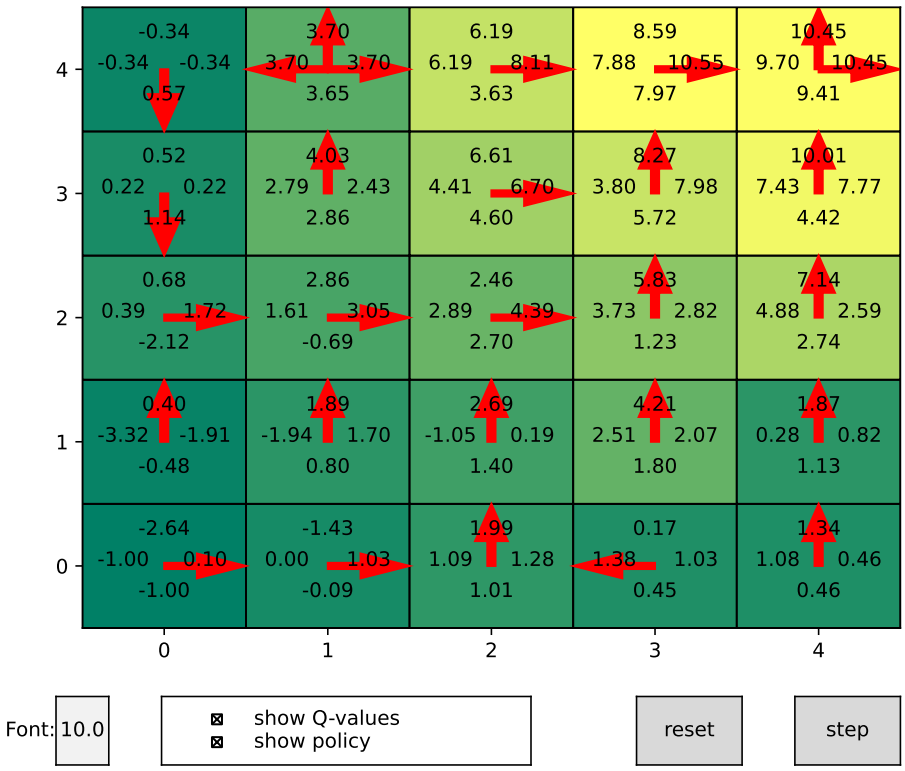


Figure 12.11: Q-values and optimal policy for the monster game, for the states where the agent is damaged and the goal is in the top-right.

```
247
248 def test_MDP(mdp, discount=0.9, eps=0.01):
249     """tests vi and avi give the same answer for a MDP class mdp
250     """
251     mdp1 = mdp(discount=discount)
252     q1,v1,pi1 = mdp1.vi(100)
253     mdp2 = mdp(discount=discount)
254     q2 = mdp2.avi(1000)
255     same = all(abs(q1[s][a]-q2[s][a]) < eps
256                    for s in mdp1.states
257                    for a in mdp1.actions)
258     assert same, "vi and avi are different:\n{q1}\n{q2}"
259     print(f"passed unit test. vi and avi gave same result for {mdp1.title}")
260
261 if __name__ == "__main__":
262     test_MDP(partyMDP)
```

Exercise 12.2 Implement value iteration that stores the V-values rather than the Q-values. Does it work better than storing Q? (What might "better" mean?)

Exercise 12.3 In asynchronous value iteration, try a number of different ways to choose the states and actions to update (e.g., sweeping through the state-action pairs, choosing them at random). Note that the best way may be to determine which states have had their Q-values changed the most, and then update the previous ones, but that is not so straightforward to implement, because you need to find those previous states.

Chapter 13

Reinforcement Learning

13.1 Representing Agents and Environments

The reinforcement learning agents and environments are instances of the general agent architecture of Section 2.1, where the percepts are (reward, state) pairs. The state here is the state of the environment, not the state of the agent. Thus this is assuming that the environment if **fully observable**.

Agents are told what actions are available to it to use, but don't initially know anything about the possible states.

- An agent implements the method `select_action` takes a (reward, state) returns the next action (and updates the state of the agent).
- An environment implements the method `do` that takes an action and returns a (reward, state) pair.

These are alternated to simulate the system. The simulation starts with the agent choosing the initial action given the state, using the method `initial_action(state)`, which typically remembers the state and returns a random action.

13.1.1 Environments

RL environments have names to make tracing easier. An environment also has a list of all of the actions that can be carried out in the environment. It is initialized with the initial state.

rlProblem.py — Representations for Reinforcement Learning

```
11 import random
12 import math
13 from display import Displayable
```

```
from agents import Agent, Environment
from utilities import select_from_dist, argmaxe, argmaxd, flip

class RL_env(Environment):
    def __init__(self, name, actions, state):
        """creates an environment given name, list of actions, and initial
            state"""
        self.name = name       # the name of the environment
        self.actions = actions # list of all actions
        self.state = state     # initial state
        self.reward = None     # last reward

    # must implement do(action)->(reward,state)
```

13.1.2 Agents

An agent initially knows what actions it can carry out (its abilities). The interactions is started by calling `initial_action`, which tells the agent what the initial state is. An agent typically remembers the state and returns an action. It has no reason to prefer one action over another, so it chooses an action at random.

rlProblem.py — (continued)

```
class RL_agent(Agent):
    """An RL_Agent
    has percepts (s, r) for some state s and real reward r
    """
    def __init__(self, actions):
       self.actions = actions

    def initial_action(self, env_state):
        """return the initial action, and remember the state and action
        Act randomly initially
        Could be overridden to initialize data structures (as the agent now
            knows about one state)
        """
        self.state = env_state
        self.action = random.choice(self.actions)
        return self.action
```

At each time step, an agent selects its next action action given the reward it received and the environment.

rlProblem.py — (continued)

```
    def select_action(self, reward, state):
        """
        Select the action given the reward and state
        Remember the action in self.action
        This implements "Act randomly" and should be overridden!
        """
```

```
        self.reward = reward
        self.action = random.choice(self.actions)
        return self.action

    def v(self, state):
        """estimate of the value of doing a best action in state.
        """
        return max(self.q(state,a) for a in self.actions)

    def q(self, state, action):
        """estimate of value of doing action in state. Should be
            overridden to be useful.
        """
        return 0
```

13.1.3 Simulating an Environment-Agent Interaction

The interaction between an agent and an environment is mediated by a simulator that calls the agent and the environment in turn. `Simulate` in this section is similar to `Simulate` of Section 2.1, except it is initialized by `agent.initial_action(state)`, and the rewards are accumulated.

rlProblem.py — (continued)

```
import matplotlib.pyplot as plt

class Simulate(Displayable):
    """simulate the interaction between the agent and the environment
    for n time steps.
    Returns a pair of the agent state and the environment state.
    """
    def __init__(self, agent, environment):
        self.agent = agent
        self.env = environment
        self.reward_history = [] # for plotting
        self.step = 0
        self.sum_rewards = 0

    def start(self):
        self.action = self.agent.initial_action(self.env.state)
        return self

    def go(self, n):
        for i in range(n):
            self.step += 1
            (reward,state) = self.env.do(self.action)
            self.display(2,f"step={self.step} reward={reward},
                state={state}")
            self.sum_rewards += reward
            self.reward_history.append(reward)
```

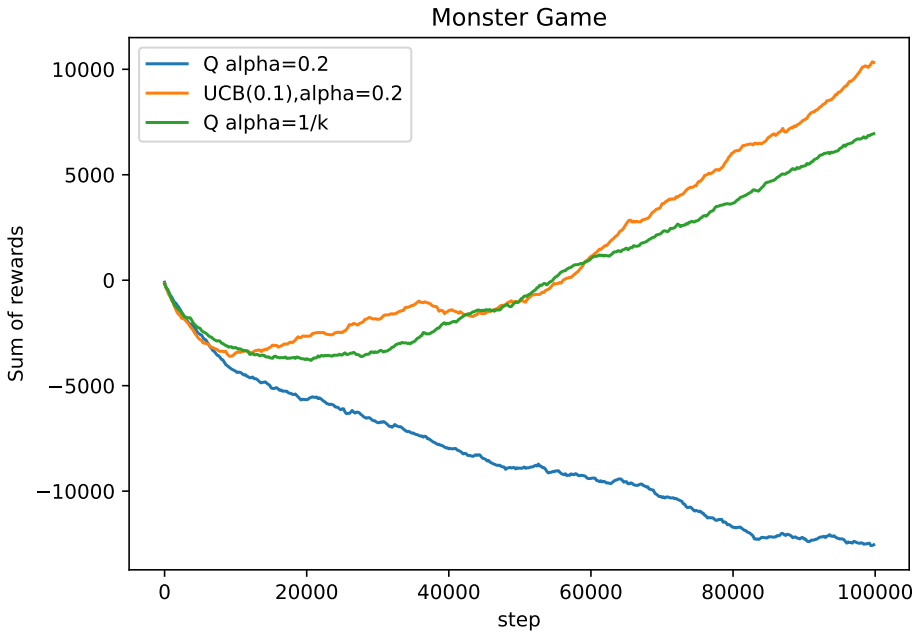


Figure 13.1: Plotting the performance of some algorithms for the monster game

```
            self.action = self.agent.select_action(reward,state)
            self.display(2,f"    action={self.action}")
        return self
```

The following plots the sum of rewards as a function of the step in a simulation. Figure 13.1 shows the performance of three algorithms for the Monster Game (Sections 12.2.1 and 13.1.6). One the x-axis is the number of actions. On the y-axis is the cumulative reward. The algorithm corresponding to the blue line has not learned very well; the plot keeps going down (but less than it did initially). The learner represented by the green line starts getting positive performance after about 20,000 steps. It took about 55,000 steps for it to have gained back the cost of exploration (when it crosses $y = 0$). The learner represented by the orange line seems to have learned quicker, but is more erratic. Each algorithm should be run multiple times, because the performance can vary a lot, even for the same problem, algorithm, and parameter settings. This graph can be reproduced (but the lines will be different) using code at the bottom of `RLQlearner.py`.

rlProblem.py — (continued)

```
    def plot(self, label=None, step_size=None, xscale='linear'):
        """
        plots the rewards history in the simulation
        label is the label for the plot
        step_size is the number of steps between each point plotted
```

```
96          xscale is 'log' or 'linear'
97
98          returns sum of rewards
99          """
100         if step_size is None: #for long simulations (> 999), only plot some
                points
101             step_size = max(1,len(self.reward_history)//500)
102         if label is None:
103             label = self.agent.name
104         plt.ion()
105         fig, ax = plt.subplots()
106         ax.set_xscale(xscale)
107         ax.set_title(self.env.name)
108         ax.set_xlabel("step")
109         ax.set_ylabel("Sum of rewards")
110         sum_history, sum_rewards = acc_rews(self.reward_history, step_size)
111         ax.plot(range(0,len(self.reward_history),step_size), sum_history,
                label=label)
112         ax.legend()
113         plt.draw()
114         return sum_rewards
115
116 def acc_rews(rews,step_size):
117     """returns the rolling sum of the values, sampled each step_size, and
            the sum
118     """
119     acc = []
120     sumr = 0; i=0
121     for e in rews:
122        sumr += e
123        i += 1
124        if (i%step_size == 0): acc.append(sumr)
125     return acc, sumr
```

13.1.4 Party Environment

Here is the definition of the simple 2-state, 2-action decision about whether to party or relax (Example 12.29 in Poole and Mackworth [2023]). (Compare to the MDP representation of page 300)

rlExamples.py — Some example reinforcement learning environments

```
11 from rlProblem import RL_env
12 class Party_env(RL_env):
13     def __init__(self):
14         RL_env.__init__(self, "Party Decision", ["party", "relax"],
               "healthy")
15
16     def do(self, action):
17         """updates the state based on the agent doing action.
18         returns reward,state
```

```
        """
        if self.state=="healthy":
            if action=="party":
                self.state = "healthy" if flip(0.7) else "sick"
                self.reward = 10
            else: # action=="relax"
                self.state = "healthy" if flip(0.95) else "sick"
                self.reward = 7
        else: # self.state=="sick"
            if action=="party":
                self.state = "healthy" if flip(0.1) else "sick"
                self.reward = 2
            else:
                self.state = "healthy" if flip(0.5) else "sick"
                self.reward = 0
        return self.reward, self.state
```

13.1.5 Environment from a Problem Domain

`Env_fom_ProblemDomain` takes a `ProblemDomain` (page 301) and constructs an environment that can be used for reinforcement learners.

As explained in Section 12.2.1, the representation of an MDP does not contain enough information to simulate a system, because it loses any dependency between the rewards and the resulting state (e.g., hitting the wall and having a negative reward may be correlated), and only represents the expected value of rewards, not how they are distributed. The `ProblemDomain` class defines the `result` method to map states and actions into distributions over (reward, state) pairs.

rlProblem.py — (continued)

```

class Env_from_ProblemDomain(RL_env):
    def __init__(self, prob_dom):
        RL_env.__init__(self, prob_dom.title, prob_dom.actions,
            prob_dom.state)
        self.problem_domain = prob_dom
        self.state = prob_dom.state
        self.x_dim = prob_dom.x_dim
        self.y_dim = prob_dom.y_dim
        self.offsets = prob_dom.offsets
        self.state2pos = self.problem_domain.state2pos
        self.state2goal = self.problem_domain.state2goal
        self.pos2state = self.problem_domain.pos2state

    def do(self, action):
        """updates the state based on the agent doing action.
        returns state,reward
        """
```

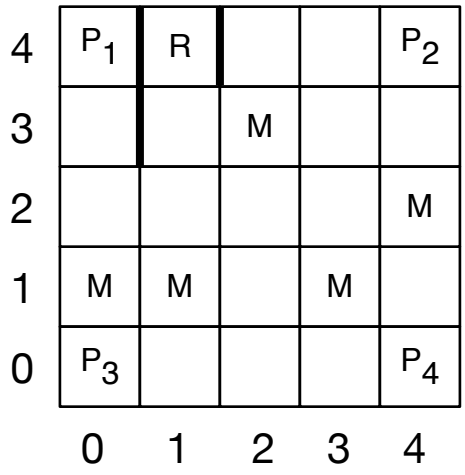


Figure 13.2: Monster game

```
144         (self.reward, self.state) =
                select_from_dist(self.problem_domain.result(self.state, action))
145         self.problem_domain.state = self.state
146         self.display(2,f"do({action} -> ({self.reward}, {self.state})")
147         return (self.reward,self.state)
```

13.1.6 Monster Game Environment

This is for the game depicted in Figure 13.2 (Example 13.2 of Poole and Mackworth [2023]). This is an alternative representation to that of Section 12.2.1, which defined the distribution over reward-state pairs. This directly builds a simulator, which might be easier to understand and easier adapt to new environments.

There are $25 * 5 * 2 = 250$ states. The agent does not know anything about how the environment works; it just knows what actions are available to it and what state it is in. It has to learn what to do.

rlExamples.py — (continued)

```
36  import random
37  from utilities import flip
38  from rlProblem import RL_env
39
40  class Monster_game_env(RL_env):
41      x_dim = 5
42      y_dim = 5
43
44      vwalls = [(0,3), (0,4), (1,4)] # vertical walls right of these locations
45      hwalls = [] # not implemented
46      crashed_reward = -1
```

```
47
48      prize_locs = [(0,0), (0,4), (4,0), (4,4)]
49      prize_apears_prob = 0.3
50      prize_reward = 10
51
52      monster_locs = [(0,1), (1,1), (2,3), (3,1), (4,2)]
53      monster_appears_prob = 0.4
54      monster_reward_when_damaged = -10
55      repair_stations = [(1,4)]
56
57      actions = ["up","down","left","right"]
58
59      def __init__(self):
60          # State:
61          self.x = 2
62          self.y = 2
63          self.damaged = False
64          self.prize = None
65          # Statistics
66          self.number_steps = 0
67          self.accumulated_rewards = 0 # sum of rewards received
68          self.min_accumulated_rewards = 0
69          self.min_step = 0
70          self.zero_crossing = 0
71          RL_env.__init__(self, "Monster Game", self.actions, (self.x,
                self.y, self.damaged, self.prize))
72          self.display(2,"","Step","Tot Rew","Ave Rew",sep="\t")
73
74      def do(self,action):
75          """updates the state based on the agent doing action.
76          returns reward,state
77          """
78          assert action in self.actions, f"Monster game, unknown action:
                {action}"
79          self.reward = 0.0
80          # A prize can appear:
81          if self.prize is None and flip(self.prize_apears_prob):
82                  self.prize = random.choice(self.prize_locs)
83          # Actions can be noisy
84          if flip(0.4):
85              actual_direction = random.choice(self.actions)
86          else:
87              actual_direction = action
88          # Modeling the actions given the actual direction
89          if actual_direction == "right":
90              if self.x==self.x_dim-1 or (self.x,self.y) in self.vwalls:
91                  self.reward += self.crashed_reward
92              else:
93                  self.x += 1
94          elif actual_direction == "left":
```

```
            if self.x==0 or (self.x-1,self.y) in self.vwalls:
                self.reward += self.crashed_reward
            else:
                self.x += -1
        elif actual_direction == "up":
            if self.y==self.y_dim-1:
                self.reward += self.crashed_reward
            else:
                self.y += 1
        elif actual_direction == "down":
            if self.y==0:
                self.reward += self.crashed_reward
            else:
                self.y += -1
        else:
            raise RuntimeError(f"unknown_direction: {actual_direction}")

        # Monsters
        if (self.x,self.y) in self.monster_locs and
            flip(self.monster_appears_prob):
            if self.damaged:
                self.reward += self.monster_reward_when_damaged
            else:
                self.damaged = True
        if (self.x,self.y) in self.repair_stations:
            self.damaged = False

        # Prizes
        if (self.x,self.y) == self.prize:
            self.reward += self.prize_reward
            self.prize = None

        # Statistics
        self.number_steps += 1
        self.accumulated_rewards += self.reward
        if self.accumulated_rewards < self.min_accumulated_rewards:
            self.min_accumulated_rewards = self.accumulated_rewards
            self.min_step = self.number_steps
        if self.accumulated_rewards>0 and
            self.reward>self.accumulated_rewards:
            self.zero_crossing = self.number_steps
        self.display(2,"",self.number_steps,self.accumulated_rewards,
                     self.accumulated_rewards/self.number_steps,sep="\t")

        return self.reward, (self.x, self.y, self.damaged, self.prize)
```

The following methods are used by the GUI (Section 13.7, page 347) so that the states can be shown.

rlExamples.py — (continued)

```
    ### For GUI
```

```
140     def state2pos(self,state):
141         """the (x,y) position for the state
142         """
143         (x, y, damaged, prize) = state
144         return (x,y)
145 
146     def state2goal(self,state):
147         """the (x,y) position for the goal
148         """
149         (x, y, damaged, prize) = state
150         return prize
151 
152     def pos2state(self,pos):
153         """the state corresponding to the (x,y) position.
154         The damages and prize are not shown in the GUI
155         """
156         (x,y) = pos
157         return (x, y, self.damaged, self.prize)
```

13.2 Q Learning

To run the Q-learning demo, in folder "aipython", load "rlQLearner.py", and copy and paste the example queries at the bottom of that file.

rlQLearner.py — Q Learning

```
11 import random
12 import math
13 from display import Displayable
14 from utilities import argmaxe, argmaxd, flip
15 from rlProblem import RL_agent, epsilon_greedy, ucb
16 
17 class Q_learner(RL_agent):
18     """A Q-learning agent has
19     belief-state consisting of
20         state is the previous state (initialized by RL_agent
21         q is a {(state,action):value} dict
22         visits is a {(state,action):n} dict. n is how many times action was
               done in state
23         acc_rewards is the accumulated reward
24     """
```

rlQLearner.py — (continued)

```
26     def __init__(self, name, actions, discount,
27                  exploration_strategy=epsilon_greedy, es_kwargs={},
28                  alpha_fun=lambda _:0.2, Qinit=0):
29         """
```

```
30        name is string representation of the agent
31        actions is the set of actions the agent can do
32        discount is the discount factor
33        exploration_strategy is the exploration function, default
              "epsilon_greedy"
34        es_kwargs is extra arguments of exploration_strategy
35        alpha_fun is a function that computes alpha from the number of
              visits
36        Qinit is the initial q-value
37        """
38        RL_agent.__init__(self, actions)
39        self.name = name
40        self.discount = discount
41        self.exploration_strategy = exploration_strategy
42        self.es_kwargs = es_kwargs
43        self.alpha_fun = alpha_fun
44        self.Qinit = Qinit
45        self.acc_rewards = 0
46        self.Q = {}
47        self.visits = {}
```

The initial action is a random action. It remembers the state, and initializes the data structures.

rlQLearner.py — (continued)

```
49    def initial_action(self, state):
50        """ Returns the initial action; selected at random
51        Initialize Data Structures
52        """
53        self.state = state
54        self.Q[state] = {act:self.Qinit for act in self.actions}
55        self.visits[state] = {act:0 for act in self.actions}
56        self.action = self.exploration_strategy(state, self.Q[state],
57                                    self.visits[state],**self.es_kwargs)
58        self.display(2, f"Initial State: {state} Action {self.action}")
59        self.display(2,"s\ta\tr\ts'\tQ")
60             # display looks best if states and actions are < 8 characters
61        return self.action
62
63    def select_action(self, reward, next_state):
64        """give reward and next state, select next action to be carried
              out"""
65        if next_state not in self.visits: # next_state not seen before
66            self.Q[next_state] = {act:self.Qinit for act in self.actions}
67            self.visits[next_state] = {act:0 for act in self.actions}
68        self.visits[self.state][self.action] +=1
69        alpha = self.alpha_fun(self.visits[self.state][self.action])
70        self.Q[self.state][self.action] += alpha*(
71                            reward
72                            + self.discount * max(self.Q[next_state].values())
73                            - self.Q[self.state][self.action])
```

```
74          self.display(2,self.state, self.action, reward, next_state,
75                       self.Q[self.state][self.action], sep='\t')
76          self.action = self.exploration_strategy(next_state,
                self.Q[next_state],
77                                      self.visits[next_state],**self.es_kwargs)
78          self.state = next_state
79          self.display(3,f"Agent {self.name} doing {self.action} in state
                {self.state}")
80          return self.action
```

The GUI requires the $q(s,a)$ functions:

rlQLearner.py — (continued)

```
82      def q(self,s,a):
83          if s in self.Q and a in self.Q[s]:
84              return self.Q[s][a]
85          else:
86              return self.Qinit
```

SARSA is the same as Q-learning except in the action selection. SARSA changes 3 lines:

rlQLearner.py — (continued)

```
88  class SARSA(Q_learner):
89      def __init__(self,*args, **nargs):
90          Q_learner.__init__(self,*args, **nargs)
91
92      def select_action(self, reward, next_state):
93          """give reward and next state, select next action to be carried
                out"""
94          if next_state not in self.visits: # next state not seen before
95              self.Q[next_state] = {act:self.Qinit for act in self.actions}
96              self.visits[next_state] = {act:0 for act in self.actions}
97          self.visits[self.state][self.action] +=1
98          alpha = self.alpha_fun(self.visits[self.state][self.action])
99          next_action = self.exploration_strategy(next_state,
                self.Q[next_state],
100                                     self.visits[next_state],**self.es_kwargs)
101         self.Q[self.state][self.action] += alpha*(
102                         reward
103                         + self.discount * self.Q[next_state][next_action]
104                         - self.Q[self.state][self.action])
105         self.display(2,self.state, self.action, reward, next_state,
106                      self.Q[self.state][self.action], sep='\t')
107         self.state = next_state
108         self.action = next_action
109         self.display(3,f"Agent {self.name} doing {self.action} in state
                {self.state}")
110         return self.action
```

13.2.1 Exploration Strategies

Two explorations strategies are defined: epsilon-greedy and upper confidence bound (UCB).

In general an exploration strategy takes two arguments, and some optional arguments depending on the strategy.

- *State* is the state that action is chosen for
- *Qs* is a $\{action : q\_value\}$ dictionary for the state
- *visits* is a $\{action : n\}$ dictionary for the current state; where n is the number of times that the action has been carried out in the current state.

rlProblem.py — (continued)

```
def epsilon_greedy(state, Qs, visits={}, epsilon=0.2):
        """select action given epsilon greedy
        Qs is the {action:Q-value} dictionary for current state
        visits is ignored
        epsilon is the probability of acting randomly
        """
        if flip(epsilon):
            return random.choice(list(Qs.keys())) # act randomly
        else:
            return argmaxd(Qs) # pick an action with max Q

def ucb(state, Qs, visits, c=1.4):
        """select action given upper-confidence bound
        Qs is the {action:Q-value} dictionary for current state
        visits is the {action:n} dictionary for current state

        0.01 is to prevent divide-by zero when visits[a]==0
        """
        Ns = sum(visits.values())
        ucb1 = {a:Qs[a]+c*math.sqrt(Ns/(0.01+visits[a]))
                    for a in Qs.keys()}
        action = argmaxd(ucb1)
        return action
```

Exercise 13.1 Implement a soft-max action selection. Choose a temperature that works well for the domain. Explain how you picked this temperature. Compare the epsilon-greedy, ucb, soft-max and optimism in the face of uncertainty for various parameter settings.

13.2.2 Testing Q-learning

The unit tests are for the 2-action 2-state decision about whether to relax or party (Example 12.29 of Poole and Mackworth [2023].

Note that simulating the same agent multiple times does not restart the agent; it keeps learning. Try the plotting some of the other methods; make sure to try multiple agents with the same parameter values before deciding whether a method with particular parameter settings is good or not. To do this, make sure you construct a new agent.

rlQLearner.py — (continued)

```
######## TEST CASES ########
from rlProblem import Simulate,epsilon_greedy, ucb, Env_from_ProblemDomain
from rlExamples import Party_env, Monster_game_env
from rlQLearner import Q_learner
from mdpExamples import MDPtiny, partyMDP

def test_RL(learnerClass, mdp=partyMDP, env=Party_env(), discount=0.9,
    eps=5, rl_steps=100000, **lkwargs):
    """tests whether RL on env has the same (within eps) Q-values as vi on
        mdp.
    eps=5 is reasonable for partyMDP (with 100000 steps) but may not be for
        other environments """
    mdp1 = mdp(discount=discount)
    q1,v1,pi1 = mdp1.vi(1000)
    ag = learnerClass(learnerClass.__name__, env.actions, discount,
        **lkwargs)
    sim = Simulate(ag,env).start()
    sim.go(rl_steps)
    same = all(abs(ag.q(s,a)-q1[s][a]) < eps
                  for s in mdp1.states
                  for a in mdp1.actions)
    assert same, (f"""Unit test failed for {env.name}, in {ag.name} Q="""
                  +str({(s,a):ag.q(s,a) for s in mdp1.states
                                        for a in mdp1.actions})
                  +f""" in vi Q={q1}""")
    print(f"Unit test passed. For {env.name}, {ag.name} has same Q-value as
        value iteration")
if __name__ == "__main__":
    test_RL(Q_learner, alpha_fun=lambda k:10/(9+k))
    #test_RL(SARSA) # should this pass? Why or why not?
```

The following are some calls you can play with. Run the commented-out code. Try other agents, including agents with the same settings.

rlQLearner.py — (continued)

```
#env = Party_env()
env = Env_from_ProblemDomain(MDPtiny())
# Some RL agents with different parameters:
ag = Q_learner("eps (0.1) greedy", env.actions, 0.7)
ag_ucb = Q_learner("ucb", env.actions, 0.7, exploration_strategy = ucb,
    es_kwargs={'c':0.1})
ag_opt = Q_learner("optimistic", env.actions, 0.7, Qinit=100,
    es_kwargs={'epsilon':0})
```

```
ag_exp_m = Q_learner("more explore", env.actions, 0.7,
    es_kwargs={'epsilon':0.5})
ag_greedy = Q_learner("disc 0.1", env.actions, 0.1, Qinit=100)
sa = SARSA("SARSA", env.actions, 0.9)
sucb = SARSA("SARSA ucb", env.actions, 0.9, exploration_strategy = ucb,
    es_kwargs={'c':1})

sim_ag = Simulate(ag,env).start()

# sim_ag.go(1000)
# ag.Q   # get the learned Q-values
# sim_ag.plot()
# sim_ucb = Simulate(ag_ucb,env).start(); sim_ucb.go(1000); sim_ucb.plot()
# Simulate(ag_opt,env).start().go(1000).plot()
# Simulate(ag_exp_m,env).start().go(1000).plot()
# Simulate(ag_greedy,env).start().go(1000).plot()
# Simulate(sa,env).start().go(1000).plot()
# Simulate(sucb,env).start().go(1000).plot()

from mdpExamples import MDPtiny
envt = Env_from_ProblemDomain(MDPtiny())
agt = Q_learner("Q alpha=0.8", envt.actions, 0.8)
#Simulate(agt, envt).start().go(1000).plot()

##### Monster Game ####
mon_env = Monster_game_env()
mag1 = Q_learner("Q alpha=0.2", mon_env.actions, 0.9)
#Simulate(mag1,mon_env).start().go(100000).plot()
mag_ucb = Q_learner("UCB(0.1),alpha=0.2", mon_env.actions, 0.9,
                    exploration_strategy = ucb, es_kwargs={'c':0.1})
#Simulate(mag_ucb,mon_env).start().go(100000).plot()

mag2 = Q_learner("Q alpha=1/k", mon_env.actions, 0.9,
                  alpha_fun=lambda k:1/k)
#Simulate(mag2,mon_env).start().go(100000).plot()
mag3 = Q_learner("alpha=10/(9+k)", mon_env.actions, 0.9,
                  alpha_fun=lambda k:10/(9+k))
#Simulate(mag3,mon_env).start().go(100000).plot()

mag4 = Q_learner("ucb & alpha=10/(9+k)", mon_env.actions, 0.9,
               alpha_fun=lambda k:10/(9+k),
               exploration_strategy = ucb, es_kwargs={'c':0.1})
#Simulate(mag4,mon_env).start().go(100000).plot()
```

13.3 Q-leaning with Experience Replay

A bounded buffer remembers values up to size `buffer_size`. Random values can be obtained using `get`. Once the bounded buffer is full, all old experiences have the same chance of being in the buffer.

rlQExperienceReplay.py — Q-Learner with Experience Replay

```
from rlQLearner import Q_learner
from utilities import flip
import random

class BoundedBuffer(object):
    def __init__(self, buffer_size=1000):
        self.buffer_size = buffer_size
        self.buffer = [0]*buffer_size
        self.number_added = 0

    def add(self, new_value):
        if self.number_added < self.buffer_size:
            self.buffer[self.number_added] = new_value
        else:
            if flip(self.buffer_size/self.number_added):
                position = random.randrange(self.buffer_size)
                self.buffer[position] = new_value
        self.number_added += 1

    def get(self):
        return self.buffer[random.randrange(min(self.number_added,
            self.buffer_size))]
```

A `Q_ER_Learner` does Q-leaning with experience replay. It only uses action replay after `burn_in` number of steps.

rlQExperienceReplay.py — (continued)

```
class Q_ER_learner(Q_learner):
    def __init__(self, name, actions, discount,
                 max_buffer_size=10000,
                 num_updates_per_action=10, burn_in=100, **q_kwargs):
        """Q-learner with experience replay
        name is the name of the agent (e.g., in a game)
        actions is the set of actions the agent can do
        discount is the discount factor
        max_buffer_size is the maximum number of past experiences that is
            remembered
        burn_in is the number of steps before using old experiences
        num_updates_per_action is the number of q-updates for past
            experiences per action
        q_kwargs are any extra parameters for Q_learner
        """
        Q_learner.__init__(self, name, actions, discount, **q_kwargs)
        self.experience_buffer = BoundedBuffer(max_buffer_size)
        self.num_updates_per_action = num_updates_per_action
        self.burn_in = burn_in

    def select_action(self, reward, next_state):
        """give reward and new state, select next action to be carried
            out"""
```

```
53        self.experience_buffer.add((self.state,self.action,reward,next_state))
              #remember experience
54        if next_state not in self.visits: # next_state not seen before
55            self.Q[next_state] = {act:self.Qinit for act in self.actions}
56            self.visits[next_state] = {act:0 for act in self.actions}
57        self.visits[self.state][self.action] +=1
58        alpha = self.alpha_fun(self.visits[self.state][self.action])
59        self.Q[self.state][self.action] += alpha*(
60                        reward
61                        + self.discount * max(self.Q[next_state].values())
62                        - self.Q[self.state][self.action])
63        self.display(2,self.state, self.action, reward, next_state,
64                     self.Q[self.state][self.action], sep='\t')
65        # do some updates from experience buffer
66        if self.experience_buffer.number_added > self.burn_in:
67            for i in range(self.num_updates_per_action):
68                (s,a,r,ns) = self.experience_buffer.get()
69                self.visits[s][a] +=1 # is this correct?
70                alpha = self.alpha_fun(self.visits[s][a])
71                self.Q[s][a] += alpha * (r +
72                                 self.discount* max(self.Q[ns][na]
73                                        for na in self.actions)
74                                 -self.Q[s][a] )
75        ### CHOOSE NEXT ACTION ###
76        self.action = self.exploration_strategy(next_state,
              self.Q[next_state],
77                                      self.visits[next_state],**self.es_kwargs)
78        self.state = next_state
79        self.display(3,f"Agent {self.name} doing {self.action} in state
              {self.state}")
80        return self.action
```

The following code plots the performance. The experience replay learner performance cannot be directly compared to Q-learning as it does more updates per action.

rlQExperienceReplay.py — (continued)

```
82  from rlProblem import Simulate
83  from rlExamples import Monster_game_env
84  from rlQLearner import mag1, mag2, mag3
85
86  mon_env = Monster_game_env()
87  mag1ar = Q_ER_learner("Q_ER", mon_env.actions,0.9,
88                        num_updates_per_action=5, burn_in=100)
89  # Simulate(mag1ar,mon_env).start().go(100000).plot()
90
91  mag3ar = Q_ER_learner("Q_ER alpha=10/(9+k)", mon_env.actions, 0.9,
92                        num_updates_per_action=50, burn_in=1000,
93                        alpha_fun=lambda k:10/(9+k))
94  # Simulate(mag3ar,mon_env).start().go(100000).plot()
95
```

```
from rlQLearner import test_RL
if __name__ == "__main__":
    test_RL(Q_ER_learner, alpha_fun=lambda k:10/(9+k))
```

Exercise 13.2 Why does this have a burn-in? What problem might this solve? How much does the burn-in affect the result?

Exercise 13.3 What is a fair way to compare the learning rate of `Q_ER_learner` and `Q_learner`, or `Q_ER_learners` with different values of `num_updates_per_action`? (Would this matter if the environment is a simulation versus in the real world?) Implement a comparison that counts the number of updates, rather than the number of actions. How much does `num_updates_per_action` matter?

13.4 Stochastic Policy Learning Agent

The following agent is like a Q-learning agent but maintains a stochastic policy. The policy is represented as unnormalized counts for each action in a state (as in a Dirichlet distribution). This is the code described in Section 14.7.2 and Figure 14.10 of Poole and Mackworth [2023].

rlStochasticPolicy.py — Simulations of agents learning

```
from display import Displayable
import utilities # argmaxall for (element,value) pairs
import matplotlib.pyplot as plt
import random
from rlQLearner import Q_learner

class StochasticPIAgent(Q_learner):
    """This agent maintains the Q-function for each state.
    Chooses the best action using empirical distribution over actions
    """
    def __init__(self, name, actions, discount=0, pi_init=1, **nargs):
        """
        name is the name of the agent (e.g., in a game)
        actions is the set of actions the agent can do.
        discount is the discount factor (0 is appropriate if there is a
            single state)
        pi_init gives the prior counts (Dirichlet prior) for the policy
            (must be >0)
        """
        #self.max_display_level = 3
        Q_learner.__init__(self, name, actions, discount,
                           exploration_strategy=self.action_from_stochastic_policy,
                           **nargs)
        self.pi_init = pi_init
        self.pi = {}

    def initial_action(self, state):
        """ update policy pi then do initial action from Q_learner
```

```
        """
        self.pi[state] = {act:self.pi_init for act in self.actions}
        return Q_learner.initial_action(self, state)

    def action_from_stochastic_policy(self, next_state, qs, vs):
        a_best = utilities.argmaxd(self.Q[self.state])
        self.pi[self.state][a_best] +=1
        if next_state not in self.pi:
            self.pi[next_state] = {act:self.pi_init for act in
                self.actions}
        return select_from_dist(self.pi[next_state])

def normalize(dist):
    """dict is a {value:number} dictionary, where the numbers are all
        non-negative
    returns dict where the numbers sum to one
    """
    tot = sum(dist.values())
    return {var:val/tot for (var,val) in dist.items()}

def select_from_dist(dist):
    rand = random.random()
    for (act,prob) in normalize(dist).items():
        rand -= prob
        if rand < 0:
            return act
```

The agent can be tested on the reinforcement learning benchmarks:

rlStochasticPolicy.py — (continued)

```
#### Testing on RL benchmarks #####
from rlProblem import Simulate
import rlExamples
mon_env = rlExamples.Monster_game_env()
magspi =StochasticPIAgent(mon_env.name, mon_env.actions,0.9)
#Simulate(magspi,mon_env).start().go(100000).plot()
magspi10 = StochasticPIAgent("stoch 10/(9+k)", mon_env.actions,0.9,
    alpha_fun=lambda k:10/(9+k))
#Simulate(magspi10,mon_env).start().go(100000).plot()

from rlQLearner import test_RL
if __name__ == "__main__":
    test_RL(StochasticPIAgent, alpha_fun=lambda k:10/(9+k))
```

Exercise 13.4 Test some other ways to determine the probabilities for the stochastic policy in `StochasticPIAgent`. (It currently can be seen as using a Dirichlet where the probability represents the proportion of times each action is best plus pseudo-counts).

Replace `self.pi[self.state][a_best] +=1` with something like `self.pi[self.state][a_best] *= c` for some $c > 1$. E.g., $c = 1.1$ so it chooses that action 10% more, independently of the number of times tried. (Try to change the

code as little as possible; make it so that either the original or different values of c can be run without changing your code. Warning: watch out for overflow.)

(a) Try for multiple c; which one works best for the Monster game?

(b) Suggest an alternative way to update the probabilities in the policy (e.g., adding δ to policy that is then normalized or some other methods). How well does it work?

13.5 Model-based Reinforcement Learner

To run the demo, in folder "`aipython`", load "`rlModelLearner.py`", and copy and paste the example queries at the bottom of that file. This assumes Python 3.

A model-based reinforcement learner builds a Markov decision process model of the domain, simultaneously learns the model and plans with that model.

The model-based reinforcement learner uses the following data structures:

- $Q[s][a]$ is dictionary that, given state s and action a returns the Q-value, the estimate of the future (discounted) value of being in state s and doing action a. (Note that `Q` is the list but `q` is the function.)
- $R[s][a]$ is dictionary that, given a (s, a) state s and action a is the average reward received from doing a in state s.
- $T[s][a][s']$ is dictionary that, given states s and s' and action a returns the number of times a was done in state s and the result was state s'. Note that s' is only a key if it has been the result of doing a in s; there are no zero counts recorded.
- $visits[s][a]$ is dictionary that, given state s and action a returns the number of times action a was carried out in state s. This is the C of Figure 13.6 of Poole and Mackworth [2023].

  Note that $visits[s][a] = \sum_{s'} T[s][a][s']$ but is stored separately to keep the code more readable.

The main difference to Figure 13.6 of Poole and Mackworth [2023] is the code below does a fixed number of asynchronous value iteration updates per step.

rlModelLearner.py — Model-based Reinforcement Learner

```
11 import random
12 from rlProblem import RL_agent, Simulate, epsilon_greedy, ucb
13 from display import Displayable
14 from utilities import argmaxe, flip
15
16 class Model_based_reinforcement_learner(RL_agent):
17     """A Model-based reinforcement learner
```

```
    """

    def __init__(self, name, actions, discount,
                 exploration_strategy=epsilon_greedy, es_kwargs={},
                 Qinit=0,
                 updates_per_step=10):
        """name is the name of the agent (e.g., in a game)
        actions is the list of actions the agent can do
        discount is the discount factor
        explore is the proportion of time the agent will explore
        Qinit is the initial value of the Q's
        updates_per_step is the number of AVI updates per action
        label is the label for plotting
        """
        RL_agent.__init__(self, actions)
        self.name = name
        self.actions = actions
        self.discount = discount
        self.exploration_strategy = exploration_strategy
        self.es_kwargs = es_kwargs
        self.Qinit = Qinit
        self.updates_per_step = updates_per_step
```

rlModelLearner.py — (continued)

```
    def initial_action(self, state):
        """ Returns the initial action; selected at random
        Initialize Data Structures

        """
        self.action = RL_agent.initial_action(self, state)
        self.T = {self.state: {a: {} for a in self.actions}}
        self.visits = {self.state: {a: 0 for a in self.actions}}
        self.Q = {self.state: {a: self.Qinit for a in self.actions}}
        self.R = {self.state: {a: 0 for a in self.actions}}
        self.states_list = [self.state] # list of states encountered
        self.display(2, f"Initial State: {state} Action {self.action}")
        self.display(2,"s\ta\tr\ts'\tQ")
        return self.action
```

rlModelLearner.py — (continued)

```
    def select_action(self, reward, next_state):
        """do num_steps of interaction with the environment
        for each action, do updates_per_step iterations of asynchronous
            value iteration
        """
        if next_state not in self.visits: # has not been encountered before
            self.states_list.append(next_state)
            self.visits[next_state] = {a:0 for a in self.actions}
            self.T[next_state] = {a:{} for a in self.actions}
            self.Q[next_state] = {a:self.Qinit for a in self.actions}
```

```
65            self.R[next_state] = {a:0 for a in self.actions}
66        if next_state in self.T[self.state][self.action]:
67            self.T[self.state][self.action][next_state] += 1
68        else:
69            self.T[self.state][self.action][next_state] = 1
70        self.visits[self.state][self.action] += 1
71        self.R[self.state][self.action] +=
              (reward-self.R[self.state][self.action])/self.visits[self.state][self.action]
72        st,act = self.state,self.action #initial state-action pair for AVI
73        for update in range(self.updates_per_step):
74            self.Q[st][act] = self.R[st][act]+self.discount*(
75                sum(self.T[st][act][nst]/self.visits[st][act]*self.v(nst)
76                    for nst in self.T[st][act].keys()))
77            st = random.choice(self.states_list)
78            act = random.choice(self.actions)
79        self.state = next_state
80        self.action = self.exploration_strategy(next_state,
              self.Q[next_state],
81                                self.visits[next_state],**self.es_kwargs)
82        return self.action
83
84    def q(self, state, action):
85        if state in self.Q and action in self.Q[state]:
86            return self.Q[state][action]
87        else:
88            return self.Qinit
```

rlModelLearner.py — (continued)

```
90  from rlExamples import Monster_game_env
91  mon_env = Monster_game_env()
92  mbl1 = Model_based_reinforcement_learner("model-based(1)",
        mon_env.actions, 0.9, updates_per_step=1)
93  # Simulate(mbl1,mon_env).start().go(100000).plot()
94  mbl10 = Model_based_reinforcement_learner("model-based(10)",
        mon_env.actions, 0.9, updates_per_step=10)
95  # Simulate(mbl10,mon_env).start().go(100000).plot()
96
97  from rlGUI import rlGUI
98  #gui = rlGUI(mon_env, mbl1)
99
100 from rlQLearner import test_RL
101 if __name__ == "__main__":
102     test_RL(Model_based_reinforcement_learner)
```

Exercise 13.5 If there were only one update per step, the algorithm could be made simpler and use less space. Explain how. Does it make it more efficient? Is it worthwhile having more than one update per step for the games implemented here?

Exercise 13.6 It is possible to implement the model-based reinforcement learner by replacing *Q*, *R*, *T*, *visits*, *res_states* with a single dictionary that, given a state

and action returns a tuple corresponding to these data structures. Does this make the algorithm easier to understand? Does this make the algorithm more efficient?

Exercise 13.7 If the states and the actions were mapped into integers, the dictionaries could be implemented perhaps more efficiently as arrays. How would the code need to change? Implement this for the monster game. Is it more efficient?

Exercise 13.8 In `random_choice` in the updates of `select_action`, all state-action pairs have the same chance of being chosen. Does selecting state-action pairs proportionally to the number of times visited work better than what is implemented? Provide evidence for your answer.

13.6 Reinforcement Learning with Features

> To run the demo, in folder "`aipython`", load "`rlFeatures.py`", and copy and paste the example queries at the bottom of that file. This assumes Python 3.

This section covers Q-learning with features, where the Q-function is a linear function of feature values.

13.6.1 Representing Features

A feature is a real-valued function from state and action. For an environment, you construct a function that takes a state and an action and returns a list (vector) of real numbers.

This code only does feature engineering: the feature set is redesigned for each problem. Deep RL uses deep learning to learn features, turns out to be trickier to get to work than is generally assumed.

`party_features3` and `party_features4` return lists of feature values for the party decision. `party_features4` has one extra feature.

rlGameFeature.py — Feature-based Reinforcement Learner

```
from rlExamples import Monster_game_env
from rlProblem import RL_env

def party_features3(state,action):
   return [1, state=="sick", action=="party"]

def party_features4(state,action):
   return [1, state=="sick", action=="party", state=="sick" and
       action=="party"]
```

Exercise 13.9 With `party_features3` what policies can be discovered? What policies cannot be represented as

The `monster_features` defines the vector of feature values for the given state and action.

rlGameFeature.py — (continued)

```
def monster_features(state,action):
    """returns the list of feature values for the state-action pair
    """
    assert action in Monster_game_env.actions, f"Monster game, unknown
        action: {action}"
    (x,y,d,p) = state
    # f1: would go to a monster
    f1 = monster_ahead(x,y,action)
    # f2: would crash into wall
    f2 = wall_ahead(x,y,action)
    # f3: action is towards a prize
    f3 = towards_prize(x,y,action,p)
    # f4: damaged and action is toward repair station
    f4 = towards_repair(x,y,action) if d else 0
    # f5: damaged and towards monster
    f5 = 1 if d and f1 else 0
    # f6: damaged
    f6 = 1 if d else 0
    # f7: not damaged
    f7 = 1-f6
    # f8: damaged and prize ahead
    f8 = 1 if d and f3 else 0
    # f9: not damaged and prize ahead
    f9 = 1 if not d and f3 else 0
    features = [1,f1,f2,f3,f4,f5,f6,f7,f8,f9]
    # the next 20 features are for 5 prize locations
    # and 4 distances from outside in all directions
    for pr in Monster_game_env.prize_locs+[None]:
        if p==pr:
            features += [x, 4-x, y, 4-y]
        else:
            features += [0, 0, 0, 0]
    # fp04 feature for y when prize is at 0,4
    # this knows about the wall to the right of the prize
    if p==(0,4):
        if x==0:
            fp04 = y
        elif y<3:
            fp04 = y
        else:
            fp04 = 4-y
    else:
        fp04 = 0
    features.append(fp04)
    return features

def monster_ahead(x,y,action):
    """returns 1 if the location expected to get to by doing
    action from (x,y) can contain a monster.
```

```
    """
    if action == "right" and (x+1,y) in Monster_game_env.monster_locs:
        return 1
    elif action == "left" and (x-1,y) in Monster_game_env.monster_locs:
        return 1
    elif action == "up" and (x,y+1) in Monster_game_env.monster_locs:
        return 1
    elif action == "down" and (x,y-1) in Monster_game_env.monster_locs:
        return 1
    else:
        return 0

def wall_ahead(x,y,action):
    """returns 1 if there is a wall in the direction of action from (x,y).
    This is complicated by the internal walls.
    """
    if action == "right" and (x==Monster_game_env.x_dim-1 or (x,y) in
        Monster_game_env.vwalls):
        return 1
    elif action == "left" and (x==0 or (x-1,y) in Monster_game_env.vwalls):
        return 1
    elif action == "up" and y==Monster_game_env.y_dim-1:
        return 1
    elif action == "down" and y==0:
        return 1
    else:
        return 0

def towards_prize(x,y,action,p):
    """action goes in the direction of the prize from (x,y)"""
    if p is None:
        return 0
    elif p==(0,4): # take into account the wall near the top-left prize
        if action == "left" and (x>1 or x==1 and y<3):
            return 1
        elif action == "down" and (x>0 and y>2):
            return 1
        elif action == "up" and (x==0 or y<2):
            return 1
        else:
            return 0
    else:
        px,py = p
        if p==(4,4) and x==0:
            if (action=="right" and y<3) or (action=="down" and y>2) or
                (action=="up" and y<2):
                return 1
            else:
                return 0
        if (action == "up" and y<py) or (action == "down" and py<y):
```

```
            return 1
        elif (action == "left" and px<x) or (action == "right" and x<px):
            return 1
        else:
            return 0

def towards_repair(x,y,action):
    """returns 1 if action is towards the repair station.
    """
    if action == "up" and (x>0 and y<4 or x==0 and y<2):
        return 1
    elif action == "left" and x>1:
        return 1
    elif action == "right" and x==0 and y<3:
        return 1
    elif action == "down" and x==0 and y>2:
        return 1
    else:
        return 0
```

The following uses a simpler set of features. In particular, it only considers whether the action will most likely result in a monster position or a wall, and whether the action moves towards the current prize.

rlGameFeature.py — (continued)

```
def simp_features(state,action):
    """returns a list of feature values for the state-action pair
    """
    assert action in Monster_game_env.actions
    (x,y,d,p) = state
    # f1: would go to a monster
    f1 = monster_ahead(x,y,action)
    # f2: would crash into wall
    f2 = wall_ahead(x,y,action)
    # f3: action is towards a prize
    f3 = towards_prize(x,y,action,p)
    return [1,f1,f2,f3]
```

13.6.2 Feature-based RL learner

This learns a linear function approximation of the Q-values. It requires the function *get_features* that given a state and an action returns a list of values for all of the features. Each environment requires this function to be provided.

rlFeatures.py — Feature-based Reinforcement Learner

```
import random
from rlProblem import RL_agent, epsilon_greedy, ucb
from display import Displayable
from utilities import argmaxe, flip
import rlGameFeature
```

```

class SARSA_LFA_learner(RL_agent):
    """A SARSA with linear function approximation (LFA) learning agent has
    """
    def __init__(self, name, actions, discount,
        get_features=rlGameFeature.party_features4,
                 exploration_strategy=epsilon_greedy, es_kwargs={},
                 step_size=0.01, winit=0):
        """name is the name of the agent (e.g., in a game)
        actions is the set of actions the agent can do
        discount is the discount factor
        get_features is a function get_features(state,action) -> list of
            feature values
        exploration_strategy is the exploration function, default
            "epsilon_greedy"
        es_kwargs is extra keyword arguments of the exploration_strategy
        step_size is gradient descent step size
        winit is the initial value of the weights
        """
        RL_agent.__init__(self, actions)
        self.name = name
        self.discount = discount
        self.exploration_strategy = exploration_strategy
        self.es_kwargs = es_kwargs
        self.get_features = get_features
        self.step_size = step_size
        self.winit = winit
```

The initial action is a random action. It remembers the state, and initializes the data structures.

rlFeatures.py — (continued)

```
    def initial_action(self, state):
        """ Returns the initial action; selected at random
        Initialize Data Structures
        """
        self.action = RL_agent.initial_action(self, state)
        self.features = self.get_features(state, self.action)
        self.weights = [self.winit for f in self.features]
        self.display(2, f"Initial State: {state} Action {self.action}")
        self.display(2,"s\ta\tr\ts'\tQ")
        return self.action
```

do takes in the number of steps.

rlFeatures.py — (continued)

```

    def q(self, state,action):
        """returns Q-value of the state and action for current weights
        """
        return dot_product(self.weights, self.get_features(state,action))

```

```
58      def select_action(self, reward, next_state):
59          """do num_steps of interaction with the environment"""
60          feature_values = self.get_features(self.state,self.action)
61          oldQ = self.q(self.state,self.action)
62          next_action = self.exploration_strategy(next_state,
                {a:self.q(next_state,a)
63                                                  for a in self.actions}, {})
64          nextQ = self.q(next_state,next_action)
65          delta = reward + self.discount * nextQ - oldQ
66          for i in range(len(self.weights)):
67              self.weights[i] += self.step_size * delta * feature_values[i]
68          self.display(2,self.state, self.action, reward, next_state,
69                       self.q(self.state,self.action), delta, sep='\t')
70          self.state = next_state
71          self.action = next_action
72          return self.action
73
74      def show_actions(self,state=None):
75          """prints the value for each action in a state.
76          This may be useful for debugging.
77          """
78          if state is None:
79              state = self.state
80          for next_act in self.actions:
81              print(next_act,dot_product(self.weights,
                    self.get_features(state,next_act)))
82
83  def dot_product(l1,l2):
84      return sum(e1*e2 for (e1,e2) in zip(l1,l2))
```

Test code:

rlFeatures.py — (continued)

```
86  from rlProblem import Simulate
87  from rlExamples import Party_env, Monster_game_env
88  import rlGameFeature
89  from rlGUI import rlGUI
90
91  party = Party_env()
92  pa3 = SARSA_LFA_learner(party.name, party.actions, 0.9,
        rlGameFeature.party_features3)
93  # Simulate(pa3,party).start().go(300).plot()
94  pa4 = SARSA_LFA_learner(party.name, party.actions, 0.9,
        rlGameFeature.party_features4)
95  # Simulate(pa4,party).start().go(300).plot()
96
97  mon_env = Monster_game_env()
98  fa1 = SARSA_LFA_learner("LFA", mon_env.actions, 0.9,
        rlGameFeature.monster_features)
99  # Simulate(fa1,mon_env).start().go(100000).plot()
```

```
fas1 = SARSA_LFA_learner("LFA (simp features)", mon_env.actions, 0.9,
    rlGameFeature.simp_features)
#Simulate(fas1,mon_env).start().go(100000).plot()
# rlGUI(mon_env, SARSA_LFA_learner(mon_env.name, mon_env.actions, 0.9,
    rlGameFeature.monster_features))

from rlQLearner import test_RL
if __name__ == "__main__":
    test_RL(SARSA_LFA_learner, es_kwargs={'epsilon':1}) # random exploration
```

Exercise 13.10 How does the step-size affect performance? Try different step sizes (e.g., 0.1, 0.001, other sizes in-between). Explain the behavior you observe. Which step size works best for this example. Explain what evidence you are basing your prediction on.

Exercise 13.11 Does having extra features always help? Does it sometime help? Does whether it helps depend on the step size? Give evidence for your claims.

Exercise 13.12 For each of the following first predict, then plot, then explain the behavior you observed:

(a) SARSA_LFA, Model-based learning (with 1 update per step) and Q-learning for 10,000 steps 20% exploring followed by 10,000 steps 100% exploiting

(b) SARSA_LFA, model-based learning and Q-learning for

  i) 100,000 steps 20% exploring followed by 100,000 steps 100% exploit

  ii) 10,000 steps 20% exploring followed by 190,000 steps 100% exploit

(c) Suppose your goal was to have the best accumulated reward after 200,000 steps. You are allowed to change the exploration rate at a fixed number of steps. For each of the methods, which is the best position to start exploiting more? Which method is better? What if you wanted to have the best reward after 10,000 or 1,000 steps?

Based on this evidence, explain when it is preferable to use SARSA_LFA, Model-based learner, or Q-learning.

Important: you need to run each algorithm more than once. Your explanation should include the variability as well as the typical behavior.

Exercise 13.13 In the call to `self.exploration_strategy`, what should the counts be? (The code above will fail for ucb, for example.) Think about the case where there are too many states. Suppose we are just learning for a neighborhood of a current state (e.g., a fixed number of steps away the from the current state); how could the algorithm be modifies to make sure it has at least explored the close neighborhood of the current state?

13.7 GUI for RL

This implements an an interactive graphical user interface for reinforcement learners. It lets the uses choose the actions and visualize the value function and/or the Q-function. It works by taking over the exploration strategy; when

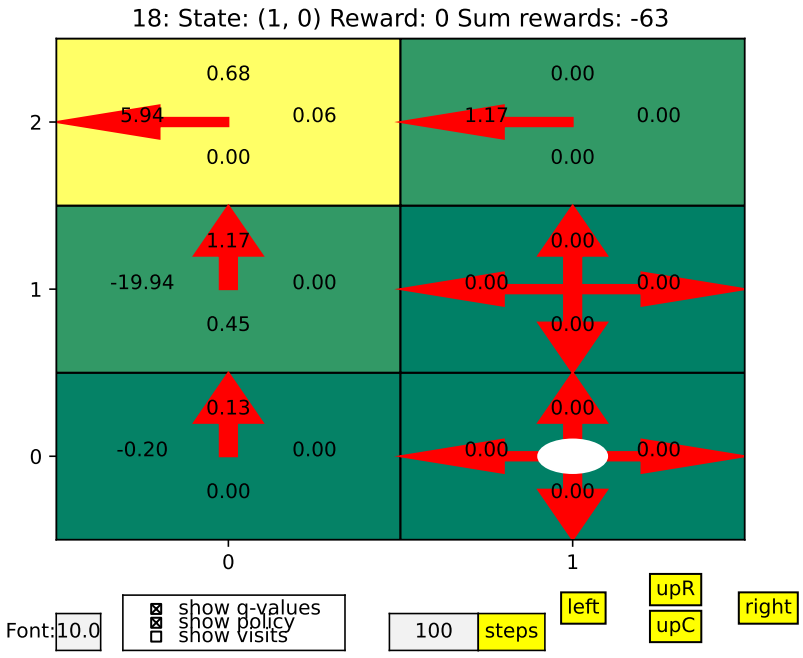


Figure 13.3: Graphical User Interface for tiny game

the agent needs to get an action, it asks the GUI. When the user requests multiple steps, it calls the original exploration strategy.

Figure 13.3 shows the GUI for the tiny game (see commented out code at the end of the file) after a 18 actions by the user. The 6 states are shown in a grid; each rectangle is a state. Within each state are 4 numbers, corresponding to the 4 actions, that give the Q-value for that state and action. The red arrows correspond to the actions with maximal Q-value for each state. The 4 yellow buttons are arranged in the same order as the Q-values. The white ellipse shows the current position of the agent. The user can simulate the agent by clicking on one of these actions. They can also click on "steps" to simulate 100 steps (in this case). The check-boxes are used to show the q-values, the policy (the red arrows) and the visits – the number of times each action has been carried out in each state (when q-values is not checked). When neither q-values or visits is checked the value for the state is shown.

Figure 13.4 shows the GUI for the monster game after 1000 steps. From the top line, you can see the agent is at location $(4, 2)$ – shown by the white dot – is damaged and the goal is at $(0, 4)$ – shown by the green dot. It is instructive to try to control the agent by clicking on the actions on the bottom right: it only does what is expected 70% of the time.

rlGUI.py — Reinforcement Learning GUI

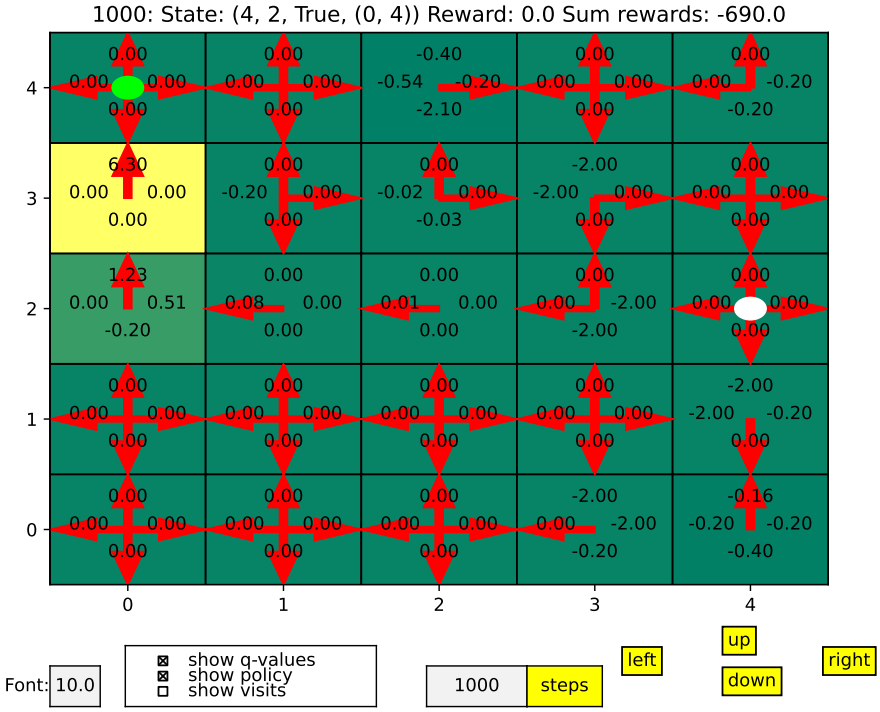


Figure 13.4: Graphical User Interface for Monster game

```
import matplotlib.pyplot as plt
from matplotlib.widgets import Button, CheckButtons, TextBox
from rlProblem import Simulate

class rlGUI(object):
    def __init__(self, env, agent):
        """
        """
        self.env = env
        self.agent = agent
        self.state = self.env.state
        self.x_dim = env.x_dim
        self.y_dim = env.y_dim
        if 'offsets' in vars(env): # 'offsets' is defined in environment
            self.offsets = env.offsets
        else: # should be more general
            self.offsets = {'right':(0.25,0), 'up':(0,0.25),
                'left':(-0.25,0), 'down':(0,-0.25)}
        # replace the exploration strategy with GUI
        self.orig_exp_strategy = self.agent.exploration_strategy
        self.agent.exploration_strategy = self.actionFromGUI
```

```
        self.do_steps = 0
        self.quitting = False
        self.action = None

    def go(self):
        self.q = self.agent.q
        self.v = self.agent.v
        try:
            self.fig,self.ax = plt.subplots()
            plt.subplots_adjust(bottom=0.2)
            self.actButtons =
                {self.fig.text(0.8+self.offsets[a][0]*0.4,0.1+self.offsets[a][1]*0.1,a,
                               bbox={'boxstyle':'square','color':'yellow','ec':'black'},
                               picker=True):a #, fontsize=fontsize):a
                 for a in self.env.actions}
            self.fig.canvas.mpl_connect('pick_event', self.sel_action)
            self.fig.canvas.mpl_connect('close_event', self.window_closed)
            self.sim = Simulate(self.agent, self.env)
            self.show()
            self.sim.start()
            self.sim.go(1000000000000) # go forever
        except ExitToPython:
            print("Window closed")

    def show(self):
        self.qcheck = CheckButtons(plt.axes([0.2,0.05,0.25,0.075]),
                                   ["show q-values","show policy","show
                                       visits"])
        self.qcheck.on_clicked(self.show_vals)
        self.font_box = TextBox(plt.axes([0.125,0.05,0.05,0.05]),"Font:",
            textalignment="center")
        self.font_box.on_submit(self.set_font_size)
        self.font_box.set_val(str(plt.rcParams['font.size']))
        self.step_box = TextBox(plt.axes([0.5,0.05,0.1,0.05]),"",
            textalignment="center")
        self.step_box.set_val("100")
        self.stepsButton = Button(plt.axes([0.6,0.05,0.075,0.05]), "steps",
            color='yellow')
        self.stepsButton.on_clicked(self.steps)
        #self.exitButton = Button(plt.axes([0.0,0.05,0.05,0.05]), "exit",
            color='yellow')
        #self.exitButton.on_clicked(self.exit)
        self.show_vals(None)

    def set_font_size(self, s):
        plt.rcParams.update({'font.size': eval(s)})
        plt.draw()

    def window_closed(self, s):
        self.quitting = True
```

```
75
76      def show_vals(self,event):
77          self.ax.cla()
78          self.ax.set_title(f"{self.sim.step}: State: {self.state} Reward:
                {self.env.reward} Sum rewards: {self.sim.sum_rewards}")
79          array = [[self.v(self.env.pos2state((x,y))) for x in
                range(self.x_dim)]
80                                              for y in range(self.y_dim)]
81          self.ax.pcolormesh([x-0.5 for x in range(self.x_dim+1)],
82                             [x-0.5 for x in range(self.y_dim+1)],
83                             array, edgecolors='black',cmap='summer')
84             # for cmap see
                   https://matplotlib.org/stable/tutorials/colors/colormaps.html
85          if self.qcheck.get_status()[1]: # "show policy"
86                 for x in range(self.x_dim):
87                     for y in range(self.y_dim):
88                        state = self.env.pos2state((x,y))
89                        maxv = max(self.agent.q(state,a) for a in
                              self.env.actions)
90                        for a in self.env.actions:
91                           xoff, yoff = self.offsets[a]
92                           if self.agent.q(state,a) == maxv:
93                              # draw arrow in appropriate direction
94                              self.ax.arrow(x,y,xoff*2,yoff*2,
95                                    color='red',width=0.05, head_width=0.2,
                                          length_includes_head=True)
96
97          if goal := self.env.state2goal(self.state):
98              self.ax.add_patch(plt.Circle(goal, 0.1, color='lime'))
99          self.ax.add_patch(plt.Circle(self.env.state2pos(self.state), 0.1,
                color='w'))
100         if self.qcheck.get_status()[0]: # "show q-values"
101            self.show_q(event)
102         elif self.qcheck.get_status()[2] and 'visits' in vars(self.agent):
                # "show visits"
103            self.show_visits(event)
104         else:
105            self.show_v(event)
106         self.ax.set_xticks(range(self.x_dim))
107         self.ax.set_xticklabels(range(self.x_dim))
108         self.ax.set_yticks(range(self.y_dim))
109         self.ax.set_yticklabels(range(self.y_dim))
110         plt.draw()
111
112     def sel_action(self,event):
113         self.action = self.actButtons[event.artist]
114
115     def show_v(self,event):
116         """show values"""
117         for x in range(self.x_dim):
```

```
118             for y in range(self.y_dim):
119                 state = self.env.pos2state((x,y))
120                 self.ax.text(x,y,"{val:.2f}".format(val=self.agent.v(state)),ha='center')
121 
122     def show_q(self,event):
123         """show q-values"""
124         for x in range(self.x_dim):
125             for y in range(self.y_dim):
126                 state = self.env.pos2state((x,y))
127                 for a in self.env.actions:
128                     xoff, yoff = self.offsets[a]
129                     self.ax.text(x+xoff,y+yoff,
130                                  "{val:.2f}".format(val=self.agent.q(state,a)),ha='center')
131 
132     def show_visits(self,event):
133         """show q-values"""
134         for x in range(self.x_dim):
135             for y in range(self.y_dim):
136                 state = self.env.pos2state((x,y))
137                 for a in self.env.actions:
138                     xoff, yoff = self.offsets[a]
139                     if state in self.agent.visits and a in
                            self.agent.visits[state]:
140                         num_visits = self.agent.visits[state][a]
141                     else:
142                         num_visits = 0
143                     self.ax.text(x+xoff,y+yoff,
144                                  str(num_visits),ha='center')
145 
146     def steps(self,event):
147         "do the steps given in step box"
148         num_steps = int(self.step_box.text)
149         if num_steps > 0:
150             self.do_steps = num_steps-1
151             self.action = self.action_from_orig_exp_strategy()
152 
153     def action_from_orig_exp_strategy(self):
154         """returns the action from the original explorations strategy"""
155         visits = self.agent.visits[self.state] if 'visits' in
                vars(self.agent) else {}
156         return
                self.orig_exp_strategy(self.state,{a:self.agent.q(self.state,a)
                for a in self.agent.actions},
157                                        visits,**self.agent.es_kwargs)
158 
159     def actionFromGUI(self, state, *args, **kwargs):
160         """called as the exploration strategy by the RL agent.
161         returns an action, either from the GUI or the original exploration
                strategy
162         """
```

```
        self.state = state
        if self.do_steps > 0: # use the original
            self.do_steps -= 1
            return self.action_from_orig_exp_strategy()
        else: # get action from the user
            self.show_vals(None)
            while self.action == None and not self.quitting: #wait for user
                action
                plt.pause(0.05) # controls reaction time of GUI
            if self.quitting:
                raise ExitToPython()
            act = self.action
            self.action = None
            return act

class ExitToPython(Exception):
    """Thrown when window closes.
    """
    pass

from rlExamples import Monster_game_env
from mdpExamples import MDPtiny, Monster_game
from rlQLearner import Q_learner, SARSA
from rlStochasticPolicy import StochasticPIAgent
from rlProblem import Env_from_ProblemDomain, epsilon_greedy, ucb

# Choose an Environment
env = Env_from_ProblemDomain(MDPtiny())
# env = Env_from_ProblemDomain(Monster_game())
# env = Monster_game_env()

# Choose an algorithm
# gui = rlGUI(env, Q_learner("Q", env.actions, 0.9)); gui.go()
# gui = rlGUI(env, SARSA("SARSA", env.actions, 0.9)); gui.go()
# gui = rlGUI(env, SARSA("SARSA alpha(k)=k:10/(9+k))", env.actions, 0.9,
    alpha_fun=lambda k:10/(9+k))); gui.go()
# gui = rlGUI(env, SARSA("SARSA-UCB", env.actions, 0.9,
    exploration_strategy = ucb, es_kwargs={'c':0.1})); gui.go()
# gui = rlGUI(env, StochasticPIAgent("Q", env.actions, 0.9,
    alpha_fun=lambda k:10/(9+k))); gui.go()

if __name__ == "__main__":
    print("Try: rlGUI(env, Q_learner('Q', env.actions, 0.9)).go()")
```

Chapter 14

Multiagent Systems

This chapter considers searching game trees and reinforcement learning for games.

14.1 Minimax

The following code implements search for two-player, zero-sum, perfect-information (fully-observable) games. One player only wins when another player loses. Such games can be modeled with

- a single value (utility) which one agent (the maximizing agent) is trying maximize and the other agent (the minimizing agent) is trying to minimize
- a game tree where the nodes correspond to state of the game (or the history of moves)
- each node is labelled by the player who controls the next move (the maximizing player or the minimizing player)
- the children of non-terminal node correspond to all of the actions by the agent controlling the node
- nodes at the end of the game have no children and are labeled with the value of the node (e.g., $+1$ for win, 0 for tie, -1 for loss).

The aim of the minimax searcher is, given a state, to find the optimal (maximizing or minimizing depending on the agent) move.

14.1.1 Creating a two-player game

masProblem.py — A Multiagent Problem

```
from display import Displayable

class Node(Displayable):
    """A node in a search tree. It has a
    name a string
    isMax is True if it is a maximizing node, otherwise it is minimizing
        node
    children is the list of children
    value is what the node evaluates to if it is a leaf.
    """
    def __init__(self, name, isMax, value, children):
        self.name = name
        self.isMax = isMax
        self.value = value
        self.allchildren = children

    def isLeaf(self):
        """returns true of this is a leaf node"""
        return self.allchildren is None

    def children(self):
        """returns the list of all children."""
        return self.allchildren

    def evaluate(self):
        """returns the evaluation for this node if it is a leaf"""
        return self.value

    def __repr__(self):
        return self.name
```

The following gives the tree of Figure 14.1 (Figure 11.5 of Poole and Mackworth [2023]); only the leaf nodes are part of the true; the other values are described Poole and Mackworth [2023, Section 14.3.1]. 888 is used as a value for those nodes without a value in the tree. (If you look at the trace of alpha-beta pruning, 888 never appears).

masProblem.py — (continued)

```
fig10_5 = Node("a",True,None, [
          Node("b",False,None, [
              Node("d",True,None, [
                  Node("h",False,None, [
                      Node("h1",True,7,None),
                      Node("h2",True,9,None)]),
                  Node("i",False,None, [
                      Node("i1",True,6,None),
                      Node("i2",True,888,None)])]),
```

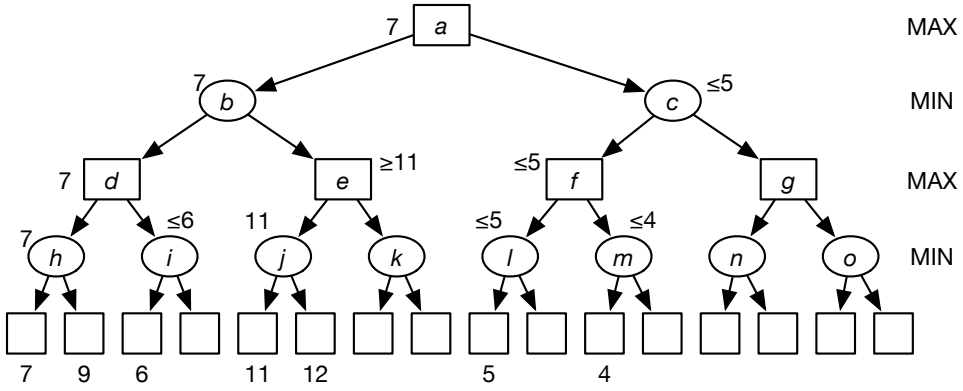


Figure 14.1: Example search tree

```
            Node("e",True,None, [
                Node("j",False,None, [
                    Node("j1",True,11,None),
                    Node("j2",True,12,None)]),
                Node("k",False,None, [
                    Node("k1",True,888,None),
                    Node("k2",True,888,None)])])]),
        Node("c",False,None, [
            Node("f",True,None, [
                Node("l",False,None, [
                    Node("l1",True,5,None),
                    Node("l2",True,888,None)]),
                Node("m",False,None, [
                    Node("m1",True,4,None),
                    Node("m2",True,888,None)])]),
            Node("g",True,None, [
                Node("n",False,None, [
                    Node("n1",True,888,None),
                    Node("n2",True,888,None)]),
                Node("o",False,None, [
                    Node("o1",True,888,None),
                    Node("o2",True,888,None)])])])])
```

The following is a representation of a **magic-sum game**, where players take turns picking a number in the range $[1, 9]$, and the first player to have 3 numbers that sum to 15 wins. Note that this is a syntactic variant of **tic-tac-toe** or **naughts and crosses**. To see this, consider the numbers on a **magic square** (Figure 14.2); 3 numbers that add to 15 correspond exactly to the winning positions of tic-tac-toe played on the magic square.

masProblem.py — (continued)

```

class Magic_sum(Node):
```

6	1	8
7	5	3
2	9	4

Figure 14.2: Magic Square

```
75      def __init__(self, xmove=True, last_move=None,
76                   available=[1,2,3,4,5,6,7,8,9], x=[], o=[]):
77          """This is a node in the search for the magic-sum game.
78          xmove is True if the next move belongs to X.
79          last_move is the number selected in the last move
80          available is the list of numbers that are available to be chosen
81          x is the list of numbers already chosen by x
82          o is the list of numbers already chosen by o
83          """
84          self.isMax = self.xmove = xmove
85          self.last_move = last_move
86          self.available = available
87          self.x = x
88          self.o = o
89          self.allchildren = None #computed on demand
90          lm = str(last_move)
91          self.name = "start" if not last_move else "o="+lm if xmove else
                "x="+lm
92
93      def children(self):
94          if self.allchildren is None:
95              if self.xmove:
96                  self.allchildren = [
97                      Magic_sum(xmove = not self.xmove,
98                                last_move = sel,
99                                available = [e for e in self.available if e is
                                    not sel],
100                               x = self.x+[sel],
101                               o = self.o)
102                             for sel in self.available]
103             else:
104                 self.allchildren = [
105                     Magic_sum(xmove = not self.xmove,
106                               last_move = sel,
107                               available = [e for e in self.available if e is
                                    not sel],
108                               x = self.x,
109                               o = self.o+[sel])
110                             for sel in self.available]
111         return self.allchildren
112
113     def isLeaf(self):
114         """A leaf has no numbers available or is a win for one of the
```

```
            players.
        We only need to check for a win for o if it is currently x's turn,
        and only check for a win for x if it is o's turn (otherwise it would
        have been a win earlier).
        """
        return (self.available == [] or
                (sum_to_15(self.last_move,self.o)
                 if self.xmove
                 else sum_to_15(self.last_move,self.x)))

    def evaluate(self):
        if self.xmove and sum_to_15(self.last_move,self.o):
            return -1
        elif not self.xmove and sum_to_15(self.last_move,self.x):
            return 1
        else:
            return 0

def sum_to_15(last,selected):
    """is true if last, together with two other elements of selected sum to
        15.
    """
    return any(last+a+b == 15
               for a in selected if a != last
               for b in selected if b != last and b != a)
```

14.1.2 Minimax and α-β Pruning

This is a naive depth-first **minimax algorithm** that searches the whole tree:

masMiniMax.py — Minimax search with alpha-beta pruning

```
def minimax(node,depth):
    """returns the value of node, and a best path for the agents
    """
    if node.isLeaf():
        return node.evaluate(),None
    elif node.isMax:
        max_score = float("-inf")
        max_path = None
        for C in node.children():
            score,path = minimax(C,depth+1)
            if score > max_score:
                max_score = score
                max_path = C.name,path
        return max_score,max_path
    else:
        min_score = float("inf")
        min_path = None
        for C in node.children():
            score,path = minimax(C,depth+1)
            if score < min_score:
```

```
31              min_score = score
32              min_path = C.name,path
33          return min_score,min_path
```

The following is a depth-first minimax with α-β **pruning**. It returns the value for a node as well as a best path for the agents.

masMiniMax.py — (continued)

```
35  def minimax_alpha_beta(node, alpha, beta, depth=0):
36      """node is a Node,
37         alpha and beta are cutoffs
38         depth is the depth on node (for indentation in printing)
39      returns value, path
40      where path is a sequence of nodes that results in the value
41      """
42      node.display(2," "*depth, f"minimax_alpha_beta({node.name}, {alpha},
            {beta})")
43      best=None     # only used if it will be pruned
44      if node.isLeaf():
45          node.display(2," "*depth, f"{node} leaf value {node.evaluate()}")
46          return node.evaluate(),None
47      elif node.isMax:
48          for C in node.children():
49              score,path = minimax_alpha_beta(C,alpha,beta,depth+1)
50              if score >= beta:  # beta pruning
51                  node.display(2," "*depth, f"{node} pruned {beta=}, {C=}")
52                  return score, None
53              if score > alpha:
54                  alpha = score
55                  best = C.name, path
56          node.display(2," "*depth, f"{node} returning max {alpha=}, {best=}")
57          return alpha,best
58      else:
59          for C in node.children():
60              score,path = minimax_alpha_beta(C,alpha,beta,depth+1)
61              if score <= alpha:   # alpha pruning
62                  node.display(2," "*depth, f"{node} pruned {alpha=}, {C=}")
63                  return score, None
64              if score < beta:
65                  beta=score
66                  best = C.name,path
67          node.display(2," "*depth, f"{node} returning min {beta=}, {best=}")
68          return beta,best
```

Testing:

masMiniMax.py — (continued)

```
70  from masProblem import fig10_5, Magic_sum, Node
71
72  # Node.max_display_level=2 # print detailed trace
73  # minimax_alpha_beta(fig10_5, -9999, 9999,0)
74  # minimax_alpha_beta(Magic_sum(), -9999, 9999,0)
```

```
75
76 #To test much time alpha-beta pruning can save over minimax:
77 ## import timeit
78 ## timeit.Timer("minimax(Magic_sum(),0)",setup="from __main__ import
       minimax, Magic_sum").timeit(number=1)
79 ## timeit.Timer("minimax_alpha_beta(Magic_sum(), -9999, 9999,0)",
       setup="from __main__ import minimax_alpha_beta,
       Magic_sum").timeit(number=1)
```

Exercise 14.1 In the magic-sum game, a state is represented as lists of moves. The same state could be reached by more than one sequence of moves. Change the representation of the game and/or the search procedures to recognize when the value of a state has already been computed. How much does this improve the search?

Exercise 14.2 There are symmetries in tic-tac toe, such as rotation and reflection. How can the representation and/or the algorithm be changed to recognize symmetries? How much difference does it make?

14.2 Multiagent Learning

The next code is for multiple agents that learn when interacting with other agents. The main difference from the simulator of the last chapter is that the games take actions from all the agents and provide a separate reward to each agent. Any of the reinforcement learning agents from the last chapter can be used.

14.2.1 Simulating Multiagent Interaction with an Environment

A game has a name, a list of player roles (which are strings for printing), a list of lists of actions (`actions[i][j]` is the `j`th action for agent `i`), a list of states, and an initial state. The default is to have a single state, and the initial state is a randomly selected state.

masLearn.py — Multiagent learning

```
11 import random
12 from display import Displayable
13 import matplotlib.pyplot as plt
14 from rlProblem import RL_agent
15
16 class Game(Displayable):
17     def __init__(self, name, players, actions, states=['s0'],
           initial_state=None):
18         self.name = name
19         self.players = players # list of roles (strings) of the players
20         self.num_players = len(players)
21         self.actions = actions # action[i] is list of actions for agent i
```

```
22        self.states = states # list of environment states; default single
              state
23        if initial_state is None:
24            self.initial_state = random.choice(states)
25        else:
26            self.initial_state = initial_state
```

The simulation for a game passes the joint action from all the agents to the environment, which returns a tuple of rewards – one for each agent – and the next state.

masLearn.py — (continued)

```
28    def sim(self, ag_types, discount=0):
29        """returns a simulation using default values for agent types
30             (This is a simple interface to SimulateGame)
31          ag_types is a list of agent functions (one for each player in the
                game)
32            The default is for one-off games where discount=0
33        """
34        return SimulateGame(self,
35                            [ag_types[i](ag_types[i].__name__,
                                self.actions[i], discount)
36                             for i in range(self.num_players)])
37
38 class SimulateGame(Displayable):
39    """A simulation of a game.
40       (This is not subclass of a game, as a game can have multiple games.)
41    """
42    def __init__(self, game, agents):
43        """ Simulates game
44            agents is a list of agents, one for each player in the game
45        """
46        #self.max_display_level = 3
47        self.game = game
48        self.agents = agents
49        # Collect Statistics:
50        self.action_counts = [{act:0 for act in game.actions[i]} for i in
              range(game.num_players)]
51        self.reward_sum = [0 for i in range(game.num_players)]
52        self.dist = {}
53        self.dist_history = []
54        self.actions = tuple(ag.initial_action(game.initial_state) for ag
              in self.agents)
55        self.num_steps = 0
56
57    def go(self, steps):
58        for i in range(steps):
59            self.num_steps += 1
60            (rewards, state) = self.game.play(self.actions)
61            self.display(3, f"In go {rewards=}, {state=}")
```

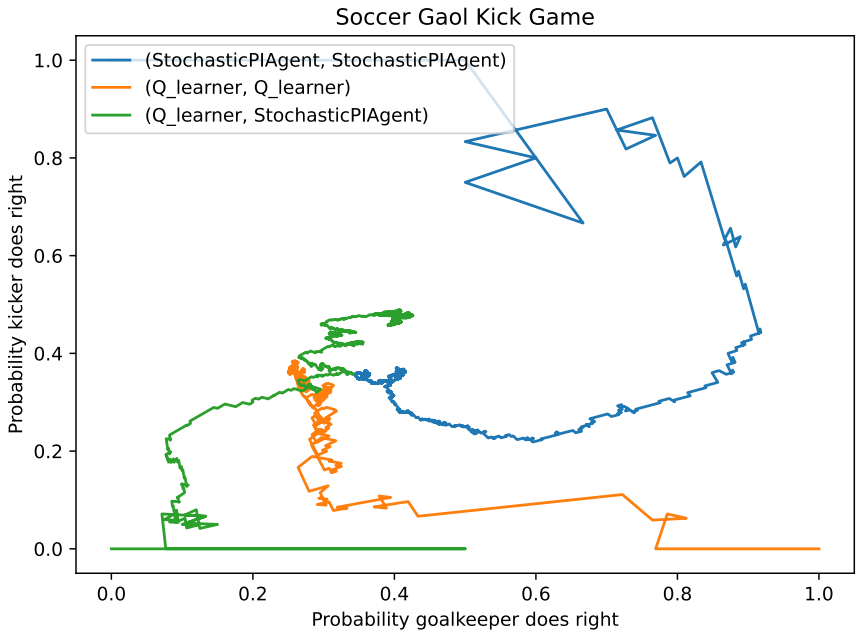


Figure 14.3: Dynamics of three runs of SoccerGame

```
        self.reward_sum = [self.reward_sum[i]+rewards[i] for i in
            range(len(rewards))]
        self.actions = tuple(agent.select_action(reward, state)
                             for (agent,reward) in
                                 zip(self.agents,rewards))
        for i in range(self.game.num_players):
            self.action_counts[i][self.actions[i]] += 1
        self.dist_history.append([{a:i/self.num_steps for (a,i) in
            elt.items()}
                                  for elt in self.action_counts])
    self.display(1,"Scores:", ' '.join(
        f"{self.agents[i].name} average
            reward={self.reward_sum[i]/self.num_steps}"
                         for i in range(self.game.num_players)))
    self.display(1,"Distributions:",
                 ' '.join(str({a:self.dist_history[-1][i][a]
                                /sum(self.dist_history[-1][i].values())
                               for a in self.game.actions[i]})
                          for  i in range(self.game.num_players)))
```

The plot shows how the empirical distributions of two actions by two agents changes as the learning continues.

Figure 14.3 shows the plot of 3 runs. The first (blue) run, where both agents are running stochastic policy iteration, starts with the goalkeeper going left

and the kicker going right; it ends with both probabilities around 0.35. The second (orange) run, where both agents are doing Q-learning, starts with the goalkeeper going right and the kicker going left; it ends with empirical probabilities of 0.24 for the goalkeeper going right and 0.36 for the kicker going right. The third (green) run, where the goalkeeper is doing Q-learning and the kicker is doing stochastic policy iteration, starts both players going left; it ends with empirical probabilities of 0.41 for the goalkeeper going right and 0.46 for the kicker going right. (You can tell the start as the empirical distribution starts with 0 or 1 probabilities, and moves quickly initially.) This figure is generated using the commented out code at the end of `masLearn.py`.

masLearn.py — (continued)

```
    def plot_dynamics(self, x_ag=0, y_ag=1, x_action=0, y_action=0):
        """ plot how the empirical probabilities vary
        x_ag index of the agent on the x-axis
        y_ag index of the agent on the y-axis
        x_action index of the action plotted for x_ag
        y_action index of the action plotted for y_ag
        """
        plt.ion() # make it interactive
        ax.set_title(self.game.name)
        x_act = self.game.actions[x_ag][x_action]
        y_act = self.game.actions[y_ag][y_action]
        ax.set_xlabel(f"Probability {self.game.players[x_ag]} does "
                      f"{self.agents[x_ag].actions[x_action]}")
        ax.set_ylabel(f"Probability {self.game.players[y_ag]} does "
                      f"{self.agents[y_ag].actions[y_action]}")
        ax.plot([self.dist_history[i][x_ag][x_act]
                    for i in range(len(self.dist_history))],
                [self.dist_history[i][y_ag][y_act]
                    for i in range(len(self.dist_history))],
                label = f"({self.agents[x_ag].name},
                    {self.agents[y_ag].name})")
        ax.legend()
        plt.show()

fig, ax = plt.subplots()
```

14.2.2 Example Games

The following are games from Poole and Mackworth [2023].

masLearn.py — (continued)

```
class ShoppingGame(Game):
    def __init__(self):
        Game.__init__(self, "Shopping Game",
                      ['football-preferrer', 'shopping-preferrer'], #players
                      [['shopping', 'football']]*2 # actions
                      )
```

```
110
111     def play(self, actions):
112         """Given (action1,action2) returns (resulting_state, (reward1,
                reward2))
113         """
114         return ({('football', 'football'): (2, 1),
115                  ('football', 'shopping'): (0, 0),
116                  ('shopping', 'football'): (0, 0),
117                  ('shopping', 'shopping'): (1, 2)
118                      }[actions], 's')
119
120 class SoccerGame(Game):
121     def __init__(self):
122         Game.__init__(self, "Soccer Gaol Kick Game",
123                         ['goalkeeper', 'kicker'], # players
124                         [['right', 'left']]*2 # actions
125                       )
126
127     def play(self, actions):
128         """Given (action1,action2) returns (resulting_state, (reward1,
                reward2))
129         resulting state is 's'
130         """
131         return ({('left', 'left'): (0.6, 0.4),
132                  ('left', 'right'): (0.3, 0.7),
133                  ('right', 'left'): (0.2, 0.8),
134                  ('right', 'right'): (0.9,0.1)
135                }[actions], 's')
136
137 class GameShow(Game):
138     def __init__(self):
139         Game.__init__(self, "Game Show (prisoners dilemma)",
140                         ['Agent 1', 'Agent 2'], # players
141                         [['takes', 'gives']]*2 # actions
142                       )
143
144     def play(self, actions):
145         return ({('takes', 'takes'): (1, 1),
146                  ('takes', 'gives'): (11, 0),
147                  ('gives', 'takes'): (0, 11),
148                  ('gives', 'gives'): (10, 10)
149                }[actions], 's')
150
151 class UniqueNEGameExample(Game):
152     def __init__(self):
153         Game.__init__(self, "3x3 Unique NE Game Example",
154                       ['agent 1', 'agent 2'], # players
155                       [['a1', 'b1', 'c1'],['d2', 'e2', 'f2']]
156                      )
157
```

```
158     def play(self, actions):
159         return ({('a1', 'd2'): (3, 5),
160                  ('a1', 'e2'): (5, 1),
161                  ('a1', 'f2'): (1, 2),
162                  ('b1', 'd2'): (1, 1),
163                  ('b1', 'e2'): (2, 9),
164                  ('b1', 'f2'): (6, 4),
165                  ('c1', 'd2'): (2, 6),
166                  ('c1', 'e2'): (4, 7),
167                  ('c1', 'f2'): (0, 8)
168                     }[actions], 's')
```

14.2.3 Testing Games and Environments

masLearn.py — (continued)

```
170 # Choose a game:
171 # gm = ShoppingGame()
172 # gm = SoccerGame()
173 # gm = GameShow()
174 # gm = UniqueNEGameExample()
175 
176 from rlQLearner import Q_learner
177 from rlProblem import RL_agent
178 from rlStochasticPolicy import StochasticPIAgent
179 # Choose one of the combinations of learners:
180 # sm = gm.sim([StochasticPIAgent, StochasticPIAgent]); sm.go(10000)
181 # sm = gm.sim([Q_learner, Q_learner]); sm.go(10000)
182 # sm = gm.sim([Q_learner, StochasticPIAgent]); sm.go(10000)
183 # sm = gm.sim([StochasticPIAgent, Q_learner]); sm.go(10000)
184 
185 # sm.plot_dynamics()
```

Exercise 14.3 Consider a pair of controllers for a games (try multiple controllers and games, including the soccer game). Does the empirical distribution represent a Nash equilibrium? Would either agent be better off if they played a Nash equilibrium instead of the empirical distribution? [10000 steps might not be enough for the algorithm to converge.]

Exercise 14.4 Try the Game Show (prisoner's dilemma) with two `StochasticPIAgent` agents and `alpha_fun=lambda k:0.1`, and also with other values of `k`, including `0.01`. Do different values of `k` work qualitatively differently? Explain why. Is one better? Try other games and other algorithms.

Exercise 14.5 Consider the alternative ways to implement stochastic policy iteration of Exercise 13.4.

(a) What value(s) of c converge for the soccer game? Explain your results.

(b) Suggest another method that works well for the soccer game, the other games and other RL environments.

Exercise 14.6 For the soccer game, how can a `Q_learner` be regularly beaten? Assume that the random number generator is secret. (Hint: can you predict what it will do?) What happens when it is played against an adversary that knows how it learns? What happens if two of these agents are played against each other? Can a `StochasticPIAgent` be defeated in the same way?

Chapter 15

Individuals and Relations

Here we implement top-down proofs for Datalog and logic programming. This is much less efficient than Prolog, which is typically implemented by compiling to an abstract machine. If you want to do serious work, we suggest using Prolog; SWI Prolog (`https://www.swi-prolog.org`) is good.

15.1 Representing Datalog and Logic Programs

The following extends the knowledge bases of Chapter 5 to include logical variables. In that chapter, atoms did not have structure and were represented as strings. Here atoms can have arguments including variables (defined below) and constants (represented by strings).

Function symbols have the same representation as atoms. To make unification simpler and to allow treating clauses as data, `Func` is defined as an abbreviation for `Atom`.

logicRelation.py — Datalog and Logic Programs

```
from display import Displayable
import logicProblem

class Var(Displayable):
    """A logical variable"""
    def __init__(self, name):
        """name"""
        self.name = name

    def __str__(self):
        return self.name
```

```
    __repr__ = __str__

    def __eq__(self, other):
        return isinstance(other,Var) and self.name == other.name
    def __hash__(self):
        return hash(self.name)

class Atom(object):
    """An atom"""
    def __init__(self, name, args):
        self.name = name
        self.args = args

    def __str__(self):
        return f"{self.name}({', '.join(str(a) for a in self.args)})"
    __repr__ = __str__

Func = Atom # same syntax is used for function symbols
```

The following extends `Clause` of Section 5.1 to include also a set of logical variables in the clause. It also allows for atoms that are strings (as in Chapter 5) and makes them into atoms.

logicRelation.py — (continued)

```
class Clause(logicProblem.Clause):
    next_index=0
    def __init__(self, head, *args, **nargs):
        if not isinstance(head, Atom):
            head = Atom(head)
        logicProblem.Clause.__init__(self, head, *args, **nargs)
        self.logical_variables = log_vars([self.head,self.body],set())

    def rename(self):
        """create a unique copy of the clause"""
        if self.logical_variables:
            sub = {v:Var(f"{v.name}_{Clause.next_index}") for v in
                self.logical_variables}
            Clause.next_index += 1
            return Clause(apply(self.head,sub),apply(self.body,sub))
        else:
           return self

def log_vars(exp, vs):
    """the union the logical variables in exp and the set vs"""
    if isinstance(exp,Var):
        return {exp}|vs
    elif isinstance(exp,Atom):
        return log_vars(exp.name, log_vars(exp.args, vs))
    elif isinstance(exp,(list,tuple)):
        for e in exp:
            vs = log_vars(e, vs)
```

```
67      return vs
```

15.2 Unification

The unification algorithm is very close to the pseudocode of Section 15.5.3 of Poole and Mackworth [2023].

logicRelation.py — (continued)

```
69  unifdisp = Var(None) # for display
70
71  def unify(t1,t2):
72      e = [(t1,t2)]
73      s = {} # empty dictionary
74      while e:
75          (a,b) = e.pop()
76          unifdisp.display(2,f"unifying{(a,b)}, e={e},s={s}")
77          if a != b:
78              if isinstance(a,Var):
79                  e = apply(e,{a:b})
80                  s = apply(s,{a:b})
81                  s[a]=b
82              elif isinstance(b,Var):
83                  e = apply(e,{b:a})
84                  s = apply(s,{b:a})
85                  s[b]=a
86              elif isinstance(a,Atom) and isinstance(b,Atom) and
                    a.name==b.name and len(a.args)==len(b.args):
87                  e += zip(a.args,b.args)
88              elif isinstance(a,(list,tuple)) and isinstance(b,(list,tuple))
                    and len(a)==len(b ):
89                  e += zip(a,b)
90              else:
91                  return False
92      return s
93
94  def apply(e,sub):
95      """e is an expression
96      sub is a {var:val} dictionary
97      returns e with all occurrence of var replaces with val"""
98      if isinstance(e,Var) and e in sub:
99          return sub[e]
100     if isinstance(e,Atom):
101         return Atom(e.name, apply(e.args,sub))
102     if isinstance(e,list):
103         return [apply(a,sub) for a in e]
104     if isinstance(e,tuple):
105         return tuple(apply(a,sub) for a in e)
106     if isinstance(e,dict):
107         return {k:apply(v,sub) for (k,v) in e.items()}
```

```
108     else:
109         return e
```

Test cases:

logicRelation.py — (continued)

```
111 ### Test cases:
112 # unifdisp.max_display_level = 2 # show trace
113 e1 = Atom('p',[Var('X'),Var('Y'),Var('Y')])
114 e2 = Atom('p',['a',Var('Z'),'b'])
115 # apply(e1,{Var('Y'):'b'})
116 # unify(e1,e2)
117 e3 = Atom('p',['a',Var('Y'),Var('Y')])
118 e4 = Atom('p',[Var('Z'),Var('Z'),'b'])
119 # unify(e3,e4)
```

15.3 Knowledge Bases

The following modifies KB of Section 5.1 so that clause indexing is only on the predicate symbol of the head of clauses.

logicRelation.py — (continued)

```
121 class KB(logicProblem.KB):
122     """A first-order knowledge base.
123       only the indexing is changed to index on name of the head."""
124 
125     def add_clause(self, c):
126         """Add clause c to clause dictionary"""
127         if c.head.name in self.atom_to_clauses:
128             self.atom_to_clauses[c.head.name].append(c)
129         else:
130             self.atom_to_clauses[c.head.name] = [c]
```

simp_KB is the simple knowledge base of Figure 15.1 of Poole and Mackworth [2023].

relnExamples.py — Relational Knowledge Base Example

```
11 from logicRelation import Var, Atom, Clause, KB
12 
13 simp_KB = KB([
14     Clause(Atom('in',['kim','r123'])),
15     Clause(Atom('part_of',['r123','cs_building'])),
16     Clause(Atom('in',[Var('X'),Var('Y')]),
17                  [Atom('part_of',[Var('Z'),Var('Y')]),
18                   Atom('in',[Var('X'),Var('Z')])])
19     ])
```

elect_KB is the relational version of the knowledge base for the electrical system of a house, as described in Example 15.11 of Poole and Mackworth [2023].

relnExamples.py — (continued)

```
# define abbreviations to make the clauses more readable:
def lit(x): return Atom('lit',[x])
def light(x): return Atom('light',[x])
def ok(x): return Atom('ok',[x])
def live(x): return Atom('live',[x])
def connected_to(x,y): return Atom('connected_to',[x,y])
def up(x): return Atom('up',[x])
def down(x): return Atom('down',[x])

L = Var('L')
W = Var('W')
W1 = Var('W1')

elect_KB = KB([
    # lit(L) is true if light L is lit.
    Clause(lit(L),
              [light(L),
               ok(L),
               live(L)]),

    # live(W) is true if W is live (i.e., current will flow through it)
    Clause(live(W),
              [connected_to(W,W1),
               live(W1)]),

    Clause(live('outside')),

    # light(L) is true if L is a light
    Clause(light('l1')),
    Clause(light('l2')),

    # connected_to(W0,W1) is true if W0 is connected to W1 such that
    # current will flow from W1 to W0.

    Clause(connected_to('l1','w0')),
    Clause(connected_to('w0','w1'),
              [ up('s2'), ok('s2')]),
    Clause(connected_to('w0','w2'),
              [ down('s2'), ok('s2')]),
    Clause(connected_to('w1','w3'),
              [ up('s1'), ok('s1')]),
    Clause(connected_to('w2','w3'),
              [ down('s1'), ok('s1')]),
    Clause(connected_to('l2','w4')),
    Clause(connected_to('w4','w3'),
              [ up('s3'), ok('s3')]),
    Clause(connected_to('p1','w3')),
    Clause(connected_to('w3','w5'),
              [ ok('cb1')]),
```

```
    Clause(connected_to('p2','w6')),
    Clause(connected_to('w6','w5'),
              [ ok('cb2')]),
    Clause(connected_to('w5','outside'),
              [ ok('outside_connection')]),

    # up(S) is true if switch S is up
    # down(S) is true if switch S is down
    Clause(down('s1')),
    Clause(up('s2')),
    Clause(up('s3')),

    # ok(L) is true if K is working. Everything is ok:
    Clause(ok(L)),
    ])
```

15.4 Top-down Proof Procedure

The top-down proof procedure is the one defined in Section 15.5.4 of Poole and Mackworth [2023] and shown in Figure 15.5. It is like `prove` defined in Section 5.3. It implements the iterator interface so that answers can be generated one at a time (or put in a list), and returns answers. To implement "choose" it loops over all alternatives and *yields* (returns one element at a time) the successful proofs.

logicRelation.py — (continued)

```
    def ask(self, query):
        """self is the current KB
        query is a list of atoms to be proved
        generates {variable:value} dictionary"""

        qvars = list(log_vars(query, set()))
        for ans in self.prove(qvars, query):
            yield {x:v for (x,v) in zip(qvars,ans)}

    def ask_all(self, query):
        """returns a list of all answers to the query given kb"""
        return list(self.ask(query))

    def ask_one(self, query):
        """returns an answer to the query given kb or None of there are no
            answers"""
        for ans in self.ask(query):
            return ans

    def prove(self, ans, ans_body, indent=""):
        """enumerates the proofs for ans_body
        ans_body is a list of atoms to be proved
```

```
153         ans is the list of values of the query variables
154         """
155         self.display(2,indent,f"(yes({ans}) <-"," & ".join(str(a) for a in
                ans_body))
156         if ans_body==[]:
157             yield ans
158         else:
159             selected, remaining = self.select_atom(ans_body)
160             if self.built_in(selected):
161                 yield from self.eval_built_in(ans, selected, remaining,
                        indent)
162             else:
163                 for chosen_clause in self.atom_to_clauses[selected.name]:
164                     clause = chosen_clause.rename() # rename variables
165                     sub = unify(selected, clause.head)
166                     if sub is not False:
167                         self.display(3,indent,"KB.prove: selected=",
                                selected, "clause=",clause,"sub=",sub)
168                         resans = apply(ans,sub)
169                         new_ans_body = apply(clause.body+remaining, sub)
170                         yield from self.prove(resans, new_ans_body, indent+"
                                ")
171
172     def select_atom(self,lst):
173         """given list of atoms, return (selected atom, remaining atoms)
174         """
175         return lst[0],lst[1:]
176
177     def built_in(self,atom):
178         return atom.name in ['lt','triple']
179
180     def eval_built_in(self,ans, selected, remaining, indent):
181         if selected.name == 'lt': # less than
182             [a1,a2] = selected.args
183             if a1 < a2:
184                 yield from self.prove(ans, remaining, indent+" ")
185         if selected.name == 'triple': # use triple store (AIFCA Ch 16)
186             yield from self.eval_triple(ans, selected, remaining, indent)
```

The unit test run when loading is the query $in(A, B)$, from `simp_KB`. It should have two answers.

relnExamples.py — (continued)

```
86  # Example Queries:
87  # simp_KB.max_display_level = 2 # show trace
88  # ask_all(simp_KB, [Atom('in',[Var('A'),Var('B')])])
89
90  A = Var('A')
91  B = Var('B')
92
93  def test_ask_all(kb=simp_KB,
```

```
                    query=[Atom('in',[A,B])],
                    res=[{ A:'kim',B:'r123'}, {A:'kim',B: 'cs_building'}]):
    ans= kb.ask_all(query)
    assert ans == res, f"ask_all({query}) gave answer {ans}"
    print("ask_all: Passed unit test")

if __name__ == "__main__":
    test_ask_all()

# elect_KB.max_display_level = 2 # show trace
# elect_KB.ask_all([light('l1')])
# elect_KB.ask_all([light('l6')])
# elect_KB.ask_all([up(Var('X'))])
# elect_KB.ask_all([connected_to('w0',W)])
# elect_KB.ask_all([connected_to('w1',W)])
# elect_KB.ask_all([connected_to(W,'w3')])
# elect_KB.ask_all([connected_to(W1,W)])
# elect_KB.ask_all([live('w6')])
# elect_KB.ask_all([live('p1')])
# elect_KB.ask_all([Atom('lit',[L])])
# elect_KB.ask_all([Atom('lit',['l2']), live('p1')])
# elect_KB.ask_all([live(L)])
```

Exercise 15.1 Implement ask-the-user similar to Section 5.3. Augment this by allowing the user to specify which instances satisfy an atom. For example, by asking the user "for what X is w1 connected to X?"; or perhaps in a more user friendly way.

15.5 Logic Program Example

The following is an append program and the query of Example 15.30 of Poole and Mackworth [2023].

```
append(nil,W,W).
append(c(A,X),Y,c(A,Z)) <-
     append(X,Y,Z).
```

The term `c(A,X)` is represented using `Atom`

In Prolog syntax:

```
append(nil,W,W).
append([A|X],Y,[A|Z]) :-
     append(X,Y,Z).
```

The value if `lst` is `[l,i,s,t]`. The query is

```
? append(F,[L],[l,i,s,t]).
```

We first define some constants and functions to make it more readable.

logicRelation.py — (continued)

```
A = Var('A')
F = Var('F')
L  =Var('L')
W = Var('W')
X = Var('X')
Y = Var('Y')
Z = Var('Z')
def cons(h,t): return Atom('cons',[h,t])
def append(a,b,c): return Atom('append',[a,b,c])

app_KB = KB([
    Clause(append('nil',W,W)),
    Clause(append(cons(A,X), Y,cons(A,Z)),
                [append(X,Y,Z)])
    ])


lst = cons('l',cons('i',cons('s',cons('t','nil'))))
# app_KB.max_display_level = 2 #show derivation
#app_KB.ask_all([append(F,cons(A,'nil'), lst)])
# Think about the expected answer before trying:
#app_KB.ask_all([append(X, Y, lst)])
#app_KB.ask_all([append(lst, lst, L), append(X, cons('s',Y), L)])
```

Chapter 16

Knowledge Graphs and Ontologies

16.1 Triple Store

A triple store provides efficient indexing for triples. For any combination of the subject-verb-object being provided or not, it can efficiently retrieve the corresponding triples. This should be comparable in speed to commercial in-memory triple stores,. It handles fewer triples, as it is not optimized for space, and only has in-memory starage. It also have fewer bells and whistles (e.g., ways to visualize triples and traverse the graph).

A triple store implements an index that covers all cases of where the subject, verb, or object are provided or not. The unspecified parts are given using Q (with value '?'). Thus, for example, `index[(Q,vrb,Q)]` is the list of triples with verb `vrb`. `index[(sub,Q,obj)` is the list of triples with subject `sub` and object `obj`.

knowledgeGraph.py — Knowledge graph triple store

```
11 from display import Displayable
12
13 class TripleStore(Displayable):
14     Q = '?' # query position
15
16     def __init__(self):
17         self.index = {}
18
19     def add(self, triple):
20         (sb,vb,ob) = triple
21         Q = self.Q    # make it easier to read
22         add_to_index(self.index, (Q,Q,Q), triple)
```

```
        add_to_index(self.index, (Q,Q,ob), triple)
        add_to_index(self.index, (Q,vb,Q), triple)
        add_to_index(self.index, (Q,vb,ob), triple)
        add_to_index(self.index, (sb,Q,Q), triple)
        add_to_index(self.index, (sb,Q,ob), triple)
        add_to_index(self.index, (sb,vb,Q), triple)
        add_to_index(self.index, triple, triple)

    def __len__(self):
        """number of triples in the triple store"""
        return len(self.index[(Q,Q,Q)])
```

The lookup method returns a list of triples that match a pattern. The pattern is a triple of the form (i,j,k) where each of i, j, and k is either "Q" or a given value; specifying whether the subject, verb, and object are provided in the query or not. `lookup((Q,Q,Q))` returns all triples. `lookup((s,v,o))` can be used to check whether the triple `(s,v,o)` is in the triple store; it returns `[]` if the triple is not in the knowledge graph, and `[(s,v,o)]` if it is.

knowledgeGraph.py — (continued)

```
    def lookup(self, query):
        """pattern is a triple of the form (i,j,k) where
           each i, j, k is either Q or a value for the
           subject, verb and object respectively.
        returns all triples with the specified non-Q vars in corresponding
            position
        """
        if query in self.index:
            return self.index[query]
        else:
            return []

def add_to_index(dict, key, value):
    if key in dict:
        dict[key].append(value)
    else:
        dict[key] = [value]
```

Here is a simple test triple store. In Wikidata Q262802 denotes the football (soccer) player Christine Sinclair, P27 is the country of citizenship, and Q16 is Canada.

knowledgeGraph.py — (continued)

```
# test cases:
sts = TripleStore() # simple triple store
Q = TripleStore.Q # makes it easier to read
sts.add(('/entity/Q262802','http://schema.org/name',"Christine Sinclair"))
sts.add(('/entity/Q262802', '/prop/direct/P27','/entity/Q16'))
sts.add(('/entity/Q16', 'http://schema.org/name', "Canada"))

# sts.lookup(('/entity/Q262802',Q,Q))
```

```
# sts.lookup((Q,'http://schema.org/name',Q))
# sts.lookup((Q,'http://schema.org/name',"Canada"))
# sts.lookup(('/entity/Q16', 'http://schema.org/name', "Canada"))
# sts.lookup(('/entity/Q262802', 'http://schema.org/name', "Canada"))
# sts.lookup((Q,Q,Q))

def test_kg(kg=sts, q=('/entity/Q262802',Q,Q),
    res=[('/entity/Q262802','http://schema.org/name',"Christine
    Sinclair"), ('/entity/Q262802', '/prop/direct/P27','/entity/Q16')]):
   """Knowledge graph unit test"""
   ans = kg.lookup(q)
   assert res==ans, f"test_kg answer {ans}"
   print("knowledge graph unit test passed")

if __name__ == "__main__":
    test_kg()
```

To read rdf files, you can use `rdflib` (`https://rdflib.readthedocs.io/en/stable/`).

The default in `load_file` is to include only English names; multiple languages can be included in the list. If the language restriction is None, all tuples are included. Converting to strings, as done here, loses information, e.g., the language associated with the literals. If you don't want to lose information, you can use `rdflib` objects, by omitting `str` in the call to `ts.add`.

knowledgeGraph.py — (continued)

```
# before using do:
# pip install rdflib

def load_file(ts, filename, language_restriction=['en']):
    import rdflib
    g = rdflib.Graph()
    g.parse(filename)
    for (s,v,o) in g:
        if language_restriction and isinstance(o,rdflib.term.Literal) and
            o._language and o._language not in language_restriction:
            pass
        else:
            ts.add((str(s),str(v),str(o)))
    print(f"{len(g)} triples read. Triple store has {len(ts)} triples.")

TripleStore.load_file = load_file

#### Test cases ####
ts = TripleStore()
#ts.load_file('http://www.wikidata.org/wiki/Special:EntityData/Q262802.nt')
q262802 ='http://www.wikidata.org/entity/Q262802'
#res=ts.lookup((q262802, 'http://www.wikidata.org/prop/P27',Q)) # country
    of citizenship
# The attributes of the object in the first answer to the above query:
```

```
#ts.lookup((res[0][2],Q,Q))
#ts.lookup((q262802, 'http://www.wikidata.org/prop/P54',Q)) # member of
    sports team
#ts.lookup((q262802,'http://schema.org/name',Q))
```

16.2 Integrating Datalog and Triple Store

The following extends the definite clause reasoner in the previous chapter to include a built-in "triple" predicate (an atom with name "triple" and three arguments). The instances of this predicate are retrieved from the triple store. This is a simplified version of what can be done with the `semweb` library of SWI Prolog (`https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/semweb.html%27)`). For anything serious, we suggest you use that. Note that the `semweb` library uses "`rdf`" as the predicate name, and Poole and Mackworth [2023] uses "prop" in Section 16.1.3 for the same predicate as "triple".

knowledgeReasoning.py — Integrating Datalog and triple store

```
from logicRelation import Var, Atom, Clause, KB, unify, apply
from knowledgeGraph import TripleStore, sts
import random

class KBT(KB):
    def __init__(self, triplestore, statements=[]):
        self.triplestore = triplestore
        KB.__init__(self, statements)

    def eval_triple(self, ans, selected, remaining, indent):
        query = selected.args
        Q = self.triplestore.Q
        pattern = tuple(Q if isinstance(e,Var) else e for e in query)
        retrieved = self.triplestore.lookup(pattern)
        self.display(3,indent,"eval_triple:
            query=",query,"pattern=",pattern,"retrieved=",retrieved)
        for tr in random.sample(retrieved,len(retrieved)):
            sub = unify(tr, query)
            self.display(3,indent,"KB.prove:
                selected=",selected,"triple=",tr,"sub=",sub)
            if sub is not False:
                yield from self.prove(apply(ans,sub), apply(remaining,sub),
                    indent+" ")

# simple test case:
kbt = KBT(sts) # sts is simple triplestore from knowledgeGraph.py
# kbt.ask_all([Atom('triple',('http://www.wikidata.org/entity/Q262802',
    Var('P'),Var('O')))])
```

The following are some larger examples from Wikidata. You must run `load_file` to load the triples related to Christine Sinclair (Q262802). Otherwise the queries won't work.

The first query is how Christine Sinclair (Q262802) is related to Portland Thorns (Q1446672) with two hops in the knowledge graph. It is asking for a P, O and $P1$ such that

$$(Q262802, P, O)\&(0, P1, Q1446672)$$

knowledgeReasoning.py — (continued)

```
O = Var('O'); O1 = Var('O1')
P = Var('P')
P1 = Var('P1')
T = Var('T')
N = Var('N')
def triple(s,v,o): return Atom('triple',[s,v,o])
def lt(a,b): return Atom('lt',[a,b])

ts = TripleStore()
kbts = KBT(ts)
#ts.load_file('http://www.wikidata.org/wiki/Special:EntityData/Q262802.nt')
q262802 ='http://www.wikidata.org/entity/Q262802'
# How is Christine Sinclair (Q262802) related to Portland Thorns
    (Q1446672) with 2 hops:
# kbts.ask_all([triple(q262802, P, O), triple(O, P1,
    'http://www.wikidata.org/entity/Q1446672') ])
```

The second is asking for the name of a team that Christine Sinclair (Q262802) played for. It is asking for a O, T and N, where O is the reified object that gives the relationship, T is the team and N is the name of the team. Informally (with variables staring with uppercase and constants in lower case) this is

$$(q262802, p54, O)\&(O, p54, T)\&(T, name, N)$$

Notice how the reified relation 'P54' (member of sports team) is represented:

knowledgeReasoning.py — (continued)

```
# What is the name of a team that Christine Sinclair played for:
# kbts.ask_one([triple(q262802, 'http://www.wikidata.org/prop/P54',O),
    triple(O,'http://www.wikidata.org/prop/statement/P54',T),
    triple(T,'http://schema.org/name',N)])
```

The third asks for the name of a team that Christine Sinclair (Q262802) played for at two different start times. It is asking for a N, $D1$ and $D2$, N is the name of the team and $D1$ and $D2$ are the start dates. In Wikidata, P54 is "member of sports team" and P580 is "start time".

knowledgeReasoning.py — (continued)

```
# The name of a team that Christine Sinclair played for at two different
    times, and the dates
def playedtwice(s,n,d0,d1): return Atom('playedtwice',[s,n,d0,d1])
S = Var('S')
N = Var('N')
```

```
D0 = Var('D0')
D1 = Var('D2')

kbts.add_clause(Clause(playedtwice(S,N,D0,D1), [
    triple(S, 'http://www.wikidata.org/prop/P54', O),
    triple(O, 'http://www.wikidata.org/prop/statement/P54', T),
    triple(S, 'http://www.wikidata.org/prop/P54', O1),
    triple(O1,'http://www.wikidata.org/prop/statement/P54', T),
    lt(O,O1), # ensure different and only generated once
    triple(T, 'http://schema.org/name', N),
    triple(O, 'http://www.wikidata.org/prop/qualifier/P580', D0),
    triple(O1, 'http://www.wikidata.org/prop/qualifier/P580', D1)
    ]))

# kbts.ask_all([playedtwice(q262802,N,D0,D1)])
```

Chapter 17

Relational Learning

17.1 Collaborative Filtering

The code here is based on the gradient descent algorithm for matrix factorization of Koren, Bell, and Volinsky [2009].

A rating set consists of training and test data, each a list of $(user, item, rating)$ tuples.

relnCollFilt.py — Latent Property-based Collaborative Filtering

```
import random
import matplotlib.pyplot as plt
import urllib.request
from learnProblem import Learner
from display import Displayable

class Rating_set(Displayable):
    """A rating contains:
    training_data: list of (user, item, rating) triples
    test_data: list of (user, item, rating) triples
    """
    def __init__(self, training_data, test_data):
        self.training_data = training_data
        self.test_data = test_data
```

The following is a representation of Examples 17.5-17.7 of Poole and Mackworth [2023]. This is a much smaller dataset than one would expect to work well.

relnCollFilt.py — (continued)

```
grades_rs = Rating_set( # 3='A', 2='B', 1='C'
    [('s1','c1',3),  # training data
     ('s2','c1',1),
```

```
    ('s1','c2',2),
    ('s2','c3',2),
    ('s3','c2',2),
    ('s4','c3',2)],
   [('s3','c4',3),  # test data
    ('s4','c4',1)])
```

A `CF_learner` does stochastic gradient descent to make a predictor of ratings for user-item pairs.

relnCollFilt.py — (continued)

```
class CF_learner(Learner):
    def __init__(self,
                 rating_set,          # a Rating_set
                 step_size = 0.01,    # gradient descent step size
                 regularization = 1.0, # L2 regularization for full dataset
                 num_properties = 10, # number of hidden properties
                 property_range = 0.02 # properties are initialized to be
                     between
                                      # -property_range and property_range
                 ):
        self.rating_set = rating_set
        self.training_data = rating_set.training_data
        self.test_data = self.rating_set.test_data
        self.step_size = step_size
        self.regularization = regularization
        self.num_properties = num_properties
        self.num_ratings = len(self.training_data)
        self.ave_rating = (sum(r for (u,i,r) in self.training_data)
                           /self.num_ratings)
        self.users = {u for (u,i,r) in self.training_data}
        self.items = {i for (u,i,r) in self.training_data}
        self.user_bias = {u:0 for u in self.users}
        self.item_bias = {i:0 for i in self.items}
        self.user_prop = {u:[random.uniform(-property_range,property_range)
                             for p in range(num_properties)]
                           for u in self.users}
        self.item_prop = {i:[random.uniform(-property_range,property_range)
                             for p in range(num_properties)]
                           for i in self.items}
        # the _delta variables are the changes internal to a batch:
        self.user_bias_delta = {u:0 for u in self.users}
        self.item_bias_delta = {i:0 for i in self.items}
        self.user_prop_delta = {u:[0 for p in range(num_properties)]
                                for u in self.users}
        self.item_prop_delta = {i:[0 for p in range(num_properties)]
                                for i in self.items}
        # zeros is used for users and items not in the training set
        self.zeros = [0 for p in range(num_properties)]
        self.epoch = 0
        self.display(1, "Predict mean:" "(Ave Abs,AveSumSq)",
```

```
75                  "training =",self.eval2string(self.training_data,
                        useMean=True),
76                  "test =",self.eval2string(self.test_data, useMean=True))
```

`prediction` returns the current prediction of a user on an item.

relnCollFilt.py — (continued)

```
78      def prediction(self,user,item):
79          """Returns prediction for this user on this item.
80          The use of .get() is to handle users or items in test set but not
                in the training set.
81          """
82          if user in self.user_bias: # user in training set
83              if item in self.item_bias: # item in training set
84                  return (self.ave_rating
85                        + self.user_bias[user]
86                        + self.item_bias[item]
87                        + sum([self.user_prop[user][p]*self.item_prop[item][p]
88                        for p in range(self.num_properties)]))
89              else: # training set contains user but not item
90                  return (self.ave_rating + self.user_bias[user])
91          elif item in self.item_bias: # training set contains item but not
                user
92              return self.ave_rating + self.item_bias[item]
93          else:
94              return self.ave_rating
```

`learn` carries out `num_epochs` epochs of stochastic gradient descent with `batch_size` giving the number of training examples in a batch. The number of epochs is approximately the average number of times each training data point is used. It is approximate because it processes the integral number of the batch size.

relnCollFilt.py — (continued)

```
96      def learn(self, num_epochs = 50, batch_size=1000):
97          """ do (approximately) num_epochs iterations through the dataset
98          batch_size is the size of each batch of stochastic gradient
                gradient descent.
99          """
100         batch_size = min(batch_size, len(self.training_data))
101         batch_per_epoch = len(self.training_data) // batch_size #
                approximate
102         num_iter = batch_per_epoch*num_epochs
103         reglz =
                self.step_size*self.regularization*batch_size/len(self.training_data)
                #regularization per batch
104
105         for i in range(num_iter):
106             if i % batch_per_epoch == 0:
107                 self.epoch += 1
108                 self.display(1,"Epoch", self.epoch, "(Ave Abs,AveSumSq)",
```

```
                        "training =",self.eval2string(self.training_data),
                        "test =",self.eval2string(self.test_data))
            # determine errors for a batch
            for (user,item,rating) in random.sample(self.training_data,
                batch_size):
                error = self.prediction(user,item) - rating
                self.user_bias_delta[user] += error
                self.item_bias_delta[item] += error
                for p in range(self.num_properties):
                    self.user_prop_delta[user][p] +=
                        error*self.item_prop[item][p]
                    self.item_prop_delta[item][p] +=
                        error*self.user_prop[user][p]
            # Update all parameters
            for user in self.users:
                self.user_bias[user] -=
                    (self.step_size*self.user_bias_delta[user]
                                    +reglz*self.user_bias[user])
                self.user_bias_delta[user] = 0
                for p in range(self.num_properties):
                    self.user_prop[user][p] -=
                        (self.step_size*self.user_prop_delta[user][p]
                                        + reglz*self.user_prop[user][p])
                    self.user_prop_delta[user][p] = 0
            for item in self.items:
                self.item_bias[item] -=
                    (self.step_size*self.item_bias_delta[item]
                                    + reglz*self.item_bias[item])
                self.item_bias_delta[item] = 0
                for p in range(self.num_properties):
                    self.item_prop[item][p] -=
                        (self.step_size*self.item_prop_delta[item][p]
                                        + reglz*self.item_prop[item][p])
                    self.item_prop_delta[item][p] = 0
```

The evaluate method evaluates current predictions on the rating set:

relnCollFilt.py — (continued)

```
    def evaluate(self, ratings, useMean=False):
        """returns (average_absolute_error, average_sum_squares_error) for
            ratings
        """
        abs_error = 0
        sumsq_error = 0
        if not ratings: return (0,0)
        for (user,item,rating) in ratings:
            prediction = self.ave_rating if useMean else
                self.prediction(user,item)
            error = prediction - rating
            abs_error += abs(error)
            sumsq_error += error * error
```

```
148         return abs_error/len(ratings), sumsq_error/len(ratings)
149
150     def eval2string(self, *args, **nargs):
151         """returns a string form of evaluate, with fewer digits
152         """
153         (abs,ssq) = self.evaluate(*args, **nargs)
154         return f"({abs:.4f}, {ssq:.4f})"
```

Let's test the code on the grades rating set:

relnCollFilt.py — (continued)

```
156 #lg = CF_learner(grades_rs,step_size = 0.1, regularization = 0.01,
        num_properties = 1)
157 #lg.learn(num_epochs = 500)
158 # lg.item_bias
159 # lg.user_bias
160 # lg.plot_property(0,plot_all=True) # can you explain why?
```

Exercise 17.1 In using `CF_learner` with `grades_rs`, does it work better with 0 properties? Is it overfitting to the data? How can overfitting be adjusted?

Exercise 17.2 Modify the code so that `self.ave_rating` is also learned. It should start as the average rating. Should it be regularized? Does it change from the initialized value? Does it work better or worse?

Exercise 17.3 With the Movielens 100K dataset and the batch size being the whole training set, what happens to the error? How can this be fixed?

Exercise 17.4 Can the regularization avoid iterating through the parameters for all users and items after a batch? Consider items that are in many batches versus those in a few or even no batches. (Warning: This is challenging to get right.)

17.1.1 Plotting

The `plot_predictions` method plots the cumulative distributions for each ground truth. Figure 17.1 shows a plot for the Movielens 100K dataset. Consider the $rating = 1$ line. The value for x is the proportion of the predictions with predicted value $\leq x$ when the ground truth has a rating of 1. Similarly for the other lines.

Figure 17.1 is for one run on the training data. What would you expected the test data to look like?

relnCollFilt.py — (continued)

```
162     def plot_predictions(self, examples="test"):
163         """
164         examples is either "test" or "training" or the actual examples
165         """
166         if examples == "test":
167             theexamples = self.test_data
168         elif examples == "training":
169             theexamples = self.training_data
```

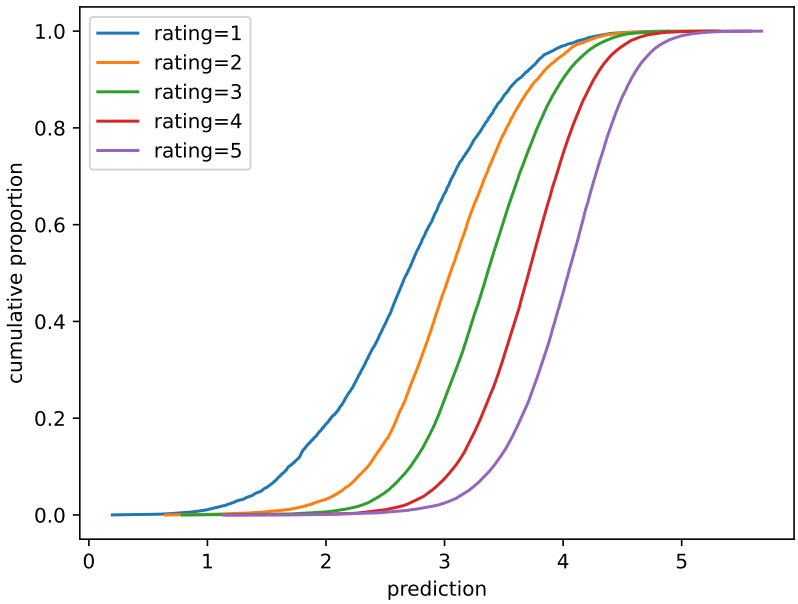


Figure 17.1: learner1.plot_predictions(examples = "training")

```
        else:
            theexamples = examples
        plt.ion()
        if not hasattr(self,'ax'):
            fig, self.ax = plt.subplots()
        self.ax.set_xlabel("prediction")
        self.ax.set_ylabel("cumulative proportion")
        self.actuals = [[] for r in range(0,6)]
        for (user,item,rating) in theexamples:
            self.actuals[rating].append(self.prediction(user,item))
        for rating in range(1,6):
            self.actuals[rating].sort()
            numrat=len(self.actuals[rating])
            yvals = [i/numrat for i in range(numrat)]
            self.ax.plot(self.actuals[rating], yvals, label=f"{examples}
                rating={rating}")
        self.ax.legend()
        plt.draw()
```

The `plot_property` method plots a single latent property; see Figure 17.2. Each $(user, item, rating)$ is plotted where the x-value is the value of the property for the user, the y-value is the value of the property for the item, and the rating is plotted at this (x, y) position. That is, *rating* is plotted at the (x, y) position $(p(user), p(item))$.

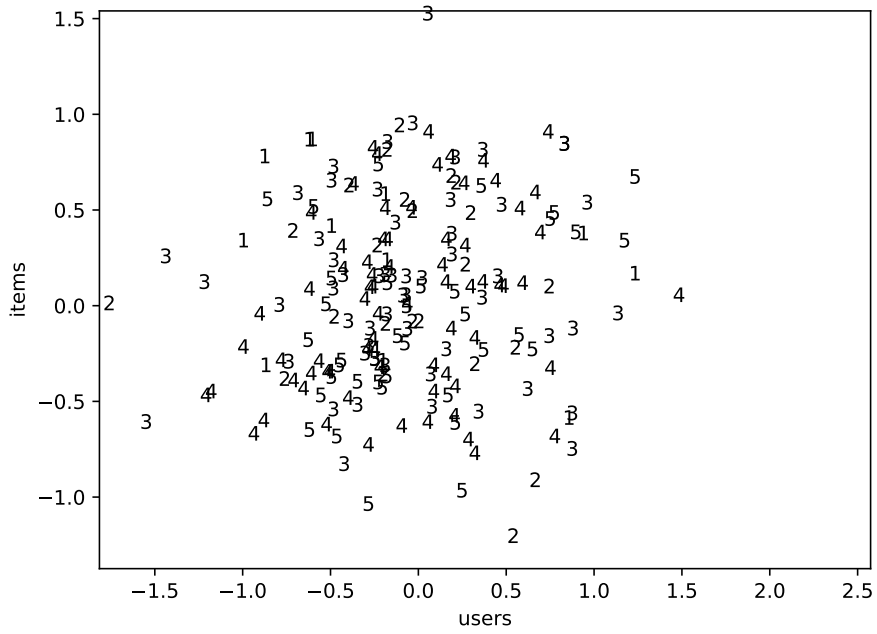


Figure 17.2: learner1.plot_property(0) with 200 random ratings plotted. Rating (u, i, r) has r plotted a position $(p(u), p(i))$ where p is the selected latent property.

Because there are too many ratings to show, `plot_property` selects a random number of points. It is difficult to see what is going on; the `create_top_subset` method was created to show the most rated items and the users who rated the most of these. This should help visualize how the latent property helps.

relnCollFilt.py — (continued)

```
188     def plot_property(self,
189                       p,              # property
190                       plot_all=False, # true if all points should be plotted
191                       num_points=200 # number of random points plotted if not
                              all
192                       ):
193         """plot some of the user-movie ratings,
194         if plot_all is true
195         num_points is the number of points selected at random plotted.
196
197         the plot has the users on the x-axis sorted by their value on
                property p and
198         with the items on the y-axis sorted by their value on property p and
199         the ratings plotted at the corresponding x-y position.
200         """
201         plt.ion()
```

```
        fig, ax = plt.subplots()
        ax.set_xlabel("users")
        ax.set_ylabel("items")
        user_vals = [self.user_prop[u][p]
                     for u in self.users]
        item_vals = [self.item_prop[i][p]
                     for i in self.items]
        ax.axis([min(user_vals)-0.02,
                 max(user_vals)+0.05,
                 min(item_vals)-0.02,
                 max(item_vals)+0.05])
        if plot_all:
            for (u,i,r) in self.training_data:
                ax.text(self.user_prop[u][p],
                        self.item_prop[i][p],
                        str(r))
        else:
            for i in range(num_points):
                (u,i,r) = random.choice(self.training_data)
                ax.text(self.user_prop[u][p],
                        self.item_prop[i][p],
                        str(r))
        plt.show()
```

17.1.2 Loading Rating Sets from Files and Websites

This assumes the form of the Movielens datasets Harper and Konstan [2015], available from `http://grouplens.org/datasets/movielens/`.

The Movielens datasets consist of $(user, movie, rating, timestamp)$ tuples. The aim here is to predict the future from the past. Tuples with a timestamp before `data_split` form the training set, and those with a timestamp after form the test set.

A rating set can be read from the Internet or read from a local file. The default is to read the Movielens 100K dataset from the Internet. It would be more efficient to save the dataset as a local file, and then set $local\_file = True$, as then it will not need to download the dataset every time the program is run.

relnCollFilt.py — (continued)

```
class Rating_set_from_file(Rating_set):
    def __init__(self,
                 date_split=892000000,
                 local_file=False,
                 url="http://files.grouplens.org/datasets/movielens/ml-100k/u.data",
                 file_name="u.data"):
        self.display(1,"Collaborative Filtering Dataset. Reading...")
        if local_file:
            lines = open(file_name,'r')
        else:
```

```
236             lines = (line.decode('utf-8') for line in
                    urllib.request.urlopen(url))
237         all_ratings = (tuple(int(e) for e in line.strip().split('\t'))
238                        for line in lines)
239         self.training_data = []
240         self.training_stats = {1:0, 2:0, 3:0, 4:0 ,5:0}
241         self.test_data = []
242         self.test_stats = {1:0, 2:0, 3:0, 4:0 ,5:0}
243         for (user,item,rating,timestamp) in all_ratings:
244             if timestamp < date_split: # rate[3] is timestamp
245                 self.training_data.append((user,item,rating))
246                 self.training_stats[rating] += 1
247             else:
248                 self.test_data.append((user,item,rating))
249                 self.test_stats[rating] += 1
250         self.display(1,"...read:", len(self.training_data),"training
                ratings and",
251                 len(self.test_data),"test ratings")
252         tr_users = {user for (user,item,rating) in self.training_data}
253         test_users = {user for (user,item,rating) in self.test_data}
254         self.display(1,"users:",len(tr_users),"training,",len(test_users),"test,",
255                      len(tr_users & test_users),"in common")
256         tr_items = {item for (user,item,rating) in self.training_data}
257         test_items = {item for (user,item,rating) in self.test_data}
258         self.display(1,"items:",len(tr_items),"training,",len(test_items),"test,",
259                      len(tr_items & test_items),"in common")
260         self.display(1,"Rating statistics for training set:
                ",self.training_stats)
261         self.display(1,"Rating statistics for test set: ",self.test_stats)
```

17.1.3 Ratings of top items and users

Sometimes it is useful to plot a property for all $(user, item, rating)$ triples. There are too many such triples in the data set. The method *create_top_subset* creates a much smaller dataset where this makes sense. It picks the most rated items, then picks the users who have the most ratings on these items. It is designed for depicting the meaning of properties, and may not be useful for other purposes. The resulting plot is shown in Figure 17.3

reInCollFilt.py — (continued)

```
263 class Rating_set_top_subset(Rating_set):
264
265     def __init__(self, rating_set, num_items = (20,40), num_users =
            (20,24)):
266         """Returns a subset of the ratings by picking the most rated items,
267         and then the users that have most ratings on these, and then all of
                the
268         ratings that involve these users and items.
```

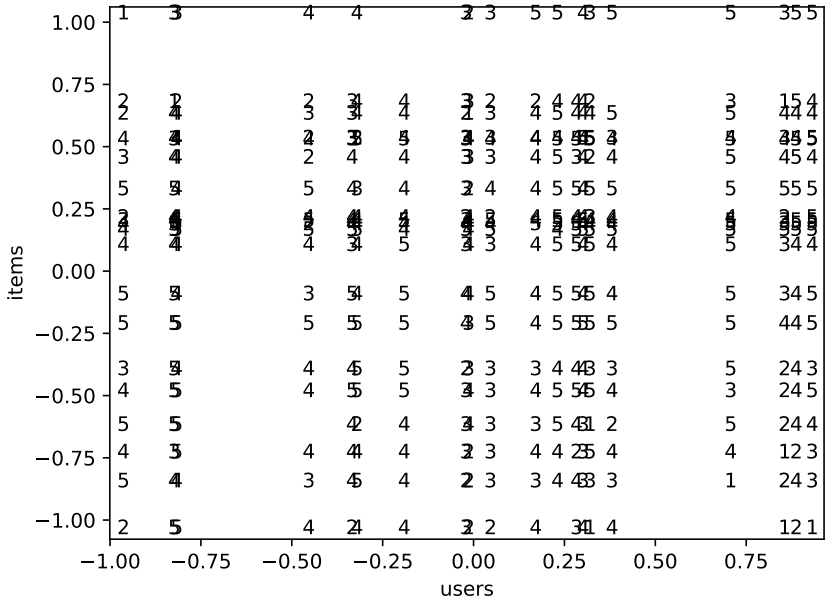


Figure 17.3: learner1.plot_property(0) for 20 most rated items and 20 users with most ratings on these. Users and items with similar property values overwrite each other.

```
        num_items is (ni,si) which selects ni users at random from the top
            si users
        num_users is (nu,su) which selects nu items at random from the top
            su items
        """
        (ni, si) = num_items
        (nu, su) = num_users
        items = {item for (user,item,rating) in rating_set.training_data}
        item_counts = {i:0 for i in items}
        for (user,item,rating) in rating_set.training_data:
            item_counts[item] += 1

        items_sorted = sorted((item_counts[i],i) for i in items)
        top_items = random.sample([item for (count, item) in
            items_sorted[-si:]], ni)
        set_top_items = set(top_items)

        users = {user for (user,item,rating) in rating_set.training_data}
        user_counts = {u:0 for u in users}
        for (user,item,rating) in rating_set.training_data:
            if item in set_top_items:
                user_counts[user] += 1
```

```
288
289         users_sorted = sorted((user_counts[u],u) for u in users)
290         top_users = random.sample([user for (count, user) in
                users_sorted[-su:]], nu)
291         set_top_users = set(top_users)
292
293         self.training_data = [ (user,item,rating)
294                         for (user,item,rating) in rating_set.training_data
295                         if user in set_top_users and item in set_top_items]
296         self.test_data = []
297
298 def test():
299     global learner1
300     movielens = Rating_set_from_file()
301     learner1 = CF_learner(movielens, num_properties = 1)
302     # learner10 = CF_learner(movielens, num_properties = 10)
303     learner1.learn(50)
304     learner1.plot_predictions(examples = "training")
305     learner1.plot_predictions(examples = "test")
306     # learner1.plot_property(0)
307     # movielens_subset = Rating_set_top_subset(movielens,num_items =
            (20,40), num_users = (20,40))
308     # learner_s = CF_learner(movielens_subset, num_properties=1)
309     # learner_s.learn(1000)
310     # learner_s.plot_property(0,plot_all=True)
311
312 if __name__ == "__main__":
313     test()
```

17.2 Relational Probabilistic Models

The following implements relational belief networks – belief networks with plates. Plates correspond to logical variables.

relnProbModels.py — Relational Probabilistic Models: belief networks with plates

```
11 from display import Displayable
12 from probGraphicalModels import BeliefNetwork
13 from variable import Variable
14 from probRC import ProbRC
15 from probFactors import Prob
16 import random
17
18 boolean = [False, True]
```

A `ParVar` is a parametrized random variable, which consists of the name, a list of logical variables (plates), a domain, and a position. For each assignment of an entity to each logical variable, there is a random variable in a grounding.

relnProbModels.py — (continued)

```
class ParVar(object):
    """Parametrized random variable"""
    def __init__(self, name, log_vars, domain, position=None):
        self.name = name # string
        self.log_vars = log_vars
        self.domain = domain # list of values
        self.position = position if position else (random.random(),
            random.random())
        self.size = len(domain)
```

The class RBN is of relational belief networks. A relational belief network consists of a title, a set of parvariables, and a set of parfactors.

relnProbModels.py — (continued)

```
class RBN(Displayable):
    def __init__(self, title, parvars, parfactors):
        self.title = title
        self.parvars = parvars
        self.parfactors = parfactors
        self.log_vars = {V for PV in parvars for V in PV.log_vars}
```

The grounding of a belief network with a population for each logical variable is a belief network, for which any of the belief network inference algorithms work.

relnProbModels.py — (continued)

```
    def ground(self, populations, offsets=None):
        """Ground the belief network with the populations of the logical
            variables.
        populations is a dictionary that maps each logical variable to the
            list of individuals.
        Returns a belief network representation of the grounding.
        """
        assert all(lv in populations for lv in self.log_vars), f"{[lv for
            lv in self.log_vars if lv not in populations]} have no
            population"
        self.cps = []   # conditional probabilities in the grounding
        self.var_dict = {}   # ground variables created
        for pp in self.parfactors:
            self.ground_parfactor(pp, list(self.log_vars), populations, {},
                offsets)
        return BeliefNetwork(self.title+"_grounded",
            self.var_dict.values(), self.cps)

    def ground_parfactor(self, parfactor, lvs, populations, context,
        offsets):
        """
        parfactor is the parfactor to get instances of
        lvs is a list of the logical variables in parfactor not assigned in
            context
        populations is {logical_variable: population} dictionary
```

```
53          context is a {logical_variable:value} dictionary for
                logical_variable in parfactor
54          offsets a {loc_var:(x_offset,y_offset)} dictionary or None
55          """
56          if lvs == []:
57              if isinstance(parfactor, Prob):
58                  self.cps.append(Prob(self.ground_pvr(parfactor.child,context,offsets),
59                                       [self.ground_pvr(p,context,offsets)
                                            for p in parfactor.parents],
60                                       parfactor.values))
61              else:
62                  print("Parfactor not implemented for",parfactor,"of
                        type",type(parfactor))
63          else:
64              for val in populations[lvs[0]]:
65                  self.ground_parfactor(parfactor, lvs[1:], populations,
                        {lvs[0]:val}|context, offsets)
66
67      def ground_pvr(self, prv, context, offsets):
68          """grounds a parametrized random variable with respect to a context
69          prv is a parametrized random variable
70          context is a logical_variable:value dictionary that assigns all
                logical variables in prv
71          offsets a {loc_var:(x_offset,y_offset)} dictionary or None
72          """
73          if isinstance(prv,ParVar):
74              args = tuple(context[lv] for lv in prv.log_vars)
75              if (prv,args) in self.var_dict:
76                  return self.var_dict[(prv,args)]
77              else:
78                  new_gv = GrVar(prv, args, offsets)
79                  self.var_dict[(prv,args)] = new_gv
80                  return new_gv
81          else: # allows for non-parametrized random variables
82              return prv
```

A `GrVar` is a variable constructed by grounding a parametrized random variable with respect to a tuple of values for the logical variables.

relnProbModels.py — (continued)

```
84  class GrVar(Variable):
85      """Grounded Variable"""
86      def __init__(self, parvar, args, offsets = None):
87          """A grounded variable
88          parvar is the parametrized variable
89          args is a tuple of a value for each random variable
90          offsets is a map between the value and the (x,y) offsets
91          """
92          if offsets:
93              pos = sum_positions([parvar.position]+[offsets[a] for a in
                    args])
```

```
        else:
           pos = sum_positions([parvar.position,
               (random.uniform(-0.2,0.2),random.uniform(-0.2,0.2))])
        Variable.__init__(self,parvar.name+"("+",".join(args)+")",
            parvar.domain, pos)
        self.parvar= parvar
        self.args = tuple(args)
        self.hash_value = None

    def __hash__(self):
        if self.hash_value is None: # only hash once
            self.hash_value = hash((self.parvar, self.args))
        return self.hash_value

    def __eq__(self, other):
        return isinstance(other,GrVar) and self.parvar == other.parvar and
            self.args == other.args

def sum_positions(poslist):
    (x,y) = (0,0)
    for (xo,yo) in poslist:
        x += xo
        y += yo
    return (x,y)
```

The following is a representation of Examples 17.5-17.7 of Poole and Mackworth [2023]. The plate model – represented here using `grades` – is shown in Figure 17.4. The observation in `obs` corresponds to the dataset of Figure 17.3. The grounding in `grades_gr` corresponds to Figure 17.5, but also includes the Grade variables not needed to answer the query (see exercise below).

Try the commented out queries to the Python shell:

relnProbModels.py — (continued)

```
Int = ParVar("Intelligent", ["St"], boolean, position=(0.0,0.7))
Grade = ParVar("Grade", ["St","Co"], ["A", "B", "C"], position=(0.2,0.6))
Diff = ParVar("Difficult", ["Co"], boolean, position=(0.3,0.9))

pg = Prob(Grade, [Int, Diff],
              [[{"A": 0.1, "B":0.4, "C":0.5},
                   {"A": 0.01, "B":0.09, "C":0.9}],
               [{"A": 0.9, "B":0.09, "C":0.01},
                    {"A": 0.5, "B":0.4, "C":0.1}]])
pi = Prob( Int, [], [0.5, 0.5])
pd = Prob( Diff, [], [0.5, 0.5])
grades = RBN("Grades RBN", {Int, Grade, Diff}, {pg,pi,pd})

students = ["s1", "s2", "s3", "s4"]
st_offsets = {st:(0,-0.2*i) for (i,st) in enumerate(students)}
courses = ["c1", "c2", "c3", "c4"]
co_offsets = {co:(0.2*i,0) for (i,co) in enumerate(courses)}
grades_gr = grades.ground({"St": students, "Co": courses},
```

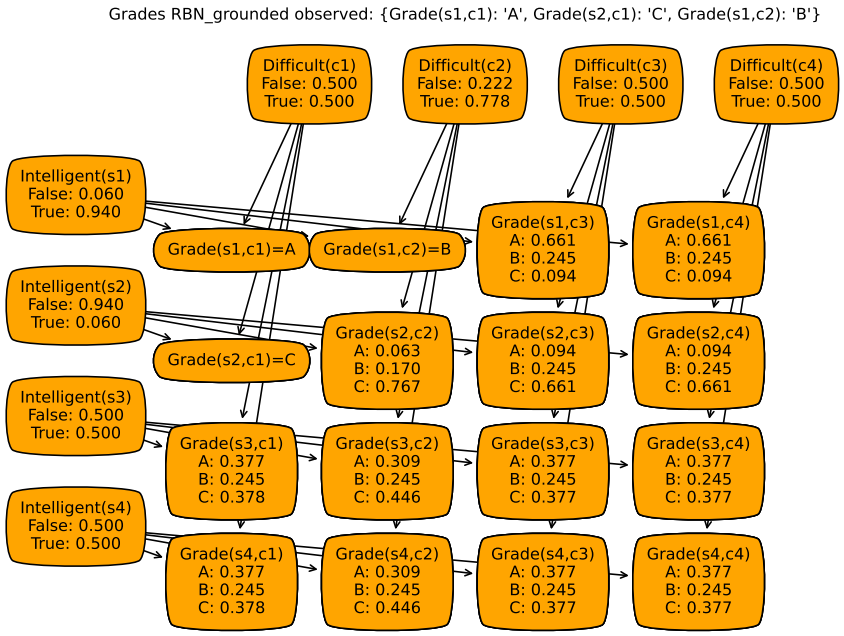


Figure 17.4: Grounded network with three observations

```
134                          offsets= st_offsets | co_offsets)
135
136 obs = {GrVar(Grade,["s1","c1"]):"A", GrVar(Grade,["s2","c1"]):"C",
        GrVar(Grade,["s1","c2"]):"B",
137           GrVar(Grade,["s2","c3"]):"B", GrVar(Grade,["s3","c2"]):"B",
                  GrVar(Grade,["s4","c3"]):"B"}
138
139 # grades_rc = ProbRC(grades_gr)
140 # grades_rc.show_post({GrVar(Grade,["s1","c1"]):"A"},fontsize=10)
141 #
        grades_rc.show_post({GrVar(Grade,["s1","c1"]):"A",GrVar(Grade,["s2","c1"]):"C"})
142 #
        grades_rc.show_post({GrVar(Grade,["s1","c1"]):"A",GrVar(Grade,["s2","c1"]):"C",
        GrVar(Grade,["s1","c2"]):"B"})
143 # grades_rc.show_post(obs,fontsize=10)
144 # grades_rc.query(GrVar(Grade,["s3","c4"]), obs)
145 # grades_rc.query(GrVar(Grade,["s4","c4"]), obs)
146 # grades_rc.query(GrVar(Int,["s3"]), obs)
147 # grades_rc.query(GrVar(Int,["s4"]), obs)
```

Figure 17.4 shows the distribution over ground variables after the 3rd `show_post` in the code above (with 3 grades observed).

Exercise 17.5 What are advantages and disadvantages of using this formulation

over using `CF_learner` with `grades_rs`? Think about overfitting, and where the parameters come from.

Exercise 17.6 The grounding above creates a random variable for each element for each possible combination of individuals in the populations. Change it so that it only creates as many random variables as needed to answer a query. For example, for the observations and queries above, only the variables in Figure 17.5 in Poole and Mackworth [2023] need to be created.

Chapter 18

Version History

- 2025-07-07 Version 0.9.17. Made it more compatible with Jupyter Notebooks by not running anything if the file is imported, and using object-oriented interface in Matplotlib. (Thanks to Jason Miller for feedback).
- 2025-04-23 Version 0.9.16. Learning and neural networks more modular. Still a candidate release for Version 1.0.
- 2024-12-19 Version 0.9.15. GUIs made more consistent and robust (with closing working).
- 2024-12-09 Version 0.9.14. Code simplified, user manual has more explanation. This is a candidate release for Version 1.0.
- 2024-04-30 Version 0.9.13: Minor changes including counterfactual reasoning.
- 2023-12-06 Version 0.9.12: Top-down proof for Datalog (ch 15) and triple store (ch 16)
- 2023-11-21 Version 0.9.11 updated and simplified relational learning, show relational belief networks
- 2023-11-07 Version 0.9.10 Improved GUIs and test cases for decision-theoretic planning (MDPs) and reinforcement learning.
- 2023-10-6 Version 0.9.8 GUIS for search, Bayesian learning, causality and many smaller changes.
- 2023-07-31 Version 0.9.7 includes relational probabilistic models and smaller changes

- 2023-06-06 Version 0.9.6 controllers are more consistent. Many smaller changes.
- 2022-08-13 Version 0.9.5 major revisions including extra code for causality and deep learning
- 2021-07-08 Version 0.9.1 updated the CSP code to have the same representation of variables as used by the probability code
- 2021-05-13 Version 0.9.0 Major revisions to chapters 8 and 9. Introduced recursive conditioning, simplified much code. New section on multiagent reinforcement learning.
- 2020-11-04 Version 0.8.6 simplified value iteration for MDPs.
- 2020-10-20 Version 0.8.4 planning simplified and fixed arc costs.
- 2020-07-21 Version 0.8.2 added positions and string to constraints
- 2019-09-17 Version 0.8.0 represented blocks world (Section 6.1.2) due to bug found by Donato Meoli.

Bibliography

Chen, T. and Guestrin, C. (2016), Xgboost: A scalable tree boosting system. In *KDD '16: 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, URL https://doi.org/10.1145/2939672.2939785. 185

Chollet, F. (2021), *Deeep Learning with Python*. Manning. 187

Dua, D. and Graff, C. (2017), UCI machine learning repository. URL http://archive.ics.uci.edu/ml. 149

Glorot, X. and Bengio, Y. (2010), Understanding the difficulty of training deep feedforward neural networks. In *Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, URL https://proceedings.mlr.press/v9/glorot10a.html. 188

Goodfellow, I., Bengio, Y., and Courville, A. (2016), *Deep Learning*. MIT Press, URL http://www.deeplearningbook.org. 195

Harper, F. M. and Konstan, J. A. (2015), The MovieLens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems*, 5(4). 392

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017), LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems 30*. 185

Koren, Y., Bell, R., and Volinsky, C. (2009), Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37. 385

Lichman, M. (2013), UCI machine learning repository. URL http://archive.ics.uci.edu/ml. 149

Pearl, J. (2009), *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2nd edition. 215, 282

Pérez, F. and Granger, B. E. (2007), IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, URL `https://ipython.org`. 10

Poole, D. L. and Mackworth, A. K. (2023), *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 3rd edition, URL `https://artint.info`. 9, 25, 27, 39, 40, 48, 50, 51, 75, 114, 123, 172, 195, 210, 213, 214, 221, 222, 265, 300, 303, 305, 306, 323, 325, 331, 336, 338, 356, 364, 371, 372, 374, 376, 382, 385, 398, 400

Index