

# Python code for Artificial Intelligence: Foundations of Computational Agents

David L. Poole and Alan K. Mackworth

Draft of April 11, 2017.

<http://artint.info>

©David L Poole and Alan K Mackworth 2017.

All code is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. See: [http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en\\_US](http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US)

This document and all the code can be downloaded from  
<http://artint.info/code/python/>.



# Contents

<b>Contents</b>	<b>3</b>
<b>1 Python for Artificial Intelligence</b>	<b>7</b>
1.1 Why Python? . . . . .	7
1.2 Getting Python . . . . .	7
1.3 Running Python . . . . .	8
1.4 Pitfalls . . . . .	9
1.5 Features of Python . . . . .	9
1.6 Useful Libraries . . . . .	13
1.7 Utilities . . . . .	14
1.8 Testing Code . . . . .	16
<b>2 Agents and Control</b>	<b>19</b>
2.1 Representing Agents and Environments . . . . .	19
2.2 Paper buying agent and environment . . . . .	20
2.3 Hierarchical Controller . . . . .	23
<b>3 Searching for Solutions</b>	<b>31</b>
3.1 Representing Search Problems . . . . .	31
3.2 A* and Related Searchers . . . . .	39
3.3 Branch-and-bound Search . . . . .	43
3.4 Depth-first Search . . . . .	45
<b>4 Features and Constraints</b>	<b>49</b>
4.1 Constraint Satisfaction Problems . . . . .	49
4.2 Solving a CSP using Search . . . . .	55

4.3	Consistency Algorithms . . . . .	56
4.4	Solving CSPs using Stochastic Local Search . . . . .	61
<b>5</b>	<b>Propositions and Inference</b>	<b>71</b>
5.1	Representing Knowledge Bases . . . . .	71
5.2	Bottom-up Proofs . . . . .	73
5.3	Top-down Proofs . . . . .	75
5.4	Assumables . . . . .	75
<b>6</b>	<b>Planning with Certainty</b>	<b>79</b>
6.1	Representing Actions and Planning Problems . . . . .	79
6.2	Forward Planner . . . . .	83
6.3	Regression Planner . . . . .	87
6.4	Planning as a CSP . . . . .	90
6.5	Partial-Order Planner . . . . .	93
<b>7</b>	<b>Learning</b>	<b>99</b>
7.1	Representations of Data and Predictions . . . . .	99
7.2	Learning With No Input Features . . . . .	109
7.3	Decision Tree Learning . . . . .	112
7.4	Cross Validation and Parameter Tuning . . . . .	116
7.5	Linear Regression and Classification . . . . .	118
7.6	Deep Neural Network Learning . . . . .	124
7.7	Boosting . . . . .	129
7.8	K-means . . . . .	131
7.9	EM . . . . .	134
<b>8</b>	<b>Reasoning Under Uncertainty</b>	<b>139</b>
8.1	Representing Probabilistic Models . . . . .	139
8.2	Factors . . . . .	140
8.3	Graphical Models . . . . .	145
8.4	Variable Elimination . . . . .	147
8.5	Stochastic Simulation . . . . .	149
8.6	Markov Chain Monte Carlo . . . . .	157
8.7	Hidden Markov Models . . . . .	159
8.8	Dynamic Belief Networks . . . . .	165
<b>9</b>	<b>Planning Under Uncertainty</b>	<b>169</b>
9.1	Decision Networks . . . . .	169
9.2	Markov Decision Processes . . . . .	173
9.3	Value Iteration . . . . .	175
<b>10</b>	<b>Multiagent Systems</b>	<b>179</b>
10.1	Minimax . . . . .	179
<b>11</b>	<b>Reinforcement Learning</b>	<b>185</b>

<i>Contents</i>	5
11.1 Representing Agents and Environments . . . . .	185
11.2 Q Learning . . . . .	191
11.3 Model-based Reinforcement Learner . . . . .	194
11.4 Reinforcement Learning with Features . . . . .	196
11.5 Learning to coordinate - UNFINISHED!!!! . . . . .	202
<b>12 Relational Learning</b>	<b>203</b>
12.1 Collaborative Filtering . . . . .	203
<b>Index</b>	<b>211</b>



## Python for Artificial Intelligence

### 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 the time, and implement just that part more efficiently in some lower-level language. Most of these 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 low-level language. You will not have to do that for the code here if you are using it for course projects.

### 1.2 Getting Python

You need Python 3 (<http://python.org/>) and matplotlib (<http://matplotlib.org/>) that runs with Python 3. This code is *not* compatible with Python 2 (e.g., with Python 2.7).

Download and install the latest Python 3 release from <http://python.org/>. This should also install *pip3*. You can install matplotlib using

```
pip3 install matplotlib
```

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

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:

```
pip3 install --upgrade matplotlib
```

We recommend using the enhanced interactive python **ipython** (<http://ipython.org/>). To install ipython after you have installed python do:

```
pip3 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<sup>1</sup>, that comes with standard Python distributions, or just running ipython3 (or perhaps just ipython) 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 following : . The first ipython3 command is in the operating system shell (note that the -i is important to enter interactive mode):

```
$ ipython3 -i searchAStar.py
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]: import searchProblem

In [2]: searcher1 = Searcher(searchProblem.acyclic_delivery_problem)

In [3]: print(searcher1.search()) # find first path
16 paths have been expanded and 5 nodes remain in the frontier
o103 --> o109 --> o119 --> o123 --> r123

In [4]: print(searcher1.search()) # find next path
21 paths have been expanded and 6 nodes remain in the frontier
o103 --> b3 --> b4 --> o109 --> o119 --> o123 --> r123

In [5]:
```

---

<sup>1</sup>See <https://hkn.eecs.berkeley.edu/~dyoo/python/idle.intro/index.html>. Note that that tutorial assumes Python 2. You will need to put the arguments to print in parentheses.



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/>. We will be using Python 3; please download the latest release. 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 will 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 happen or what may have happened. 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 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 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>

One of the nice features of Python is the use of list comprehensions (and also tuple, set and dictionary comprehensions).

$(fe \text{ for } e \text{ in } iter \text{ if } cond)$

enumerates the 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  $e$  has to be a variable,  $iter$  is an iterator, which can generate a stream of data, such as a list, a set, a range object 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.

This can go in a list, but can be called directly using *next*. The following shows a simple example, where user input is prepended with `>>>`

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

Comprehensions can also be used for dictionaries. The following code 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 3rd element of the list.

The assignment of *ind* could have also be written as:

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

where *enumerate* returns an iterator of (*index*, *value*) pairs.

### 1.5.2 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. A function uses the last value of a 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, can be easily implemented.

Consider the following programs designed to create a list of 5 functions, where the  $i$ th function in the list is meant to add  $i$  to its argument:<sup>2</sup>

```

pythonDemo.py — Some tricky examples
11 fun_list1 = []
12 for i in range(5):
13     def fun1(e):
14         return e+i
15     fun_list1.append(fun1)
16
17 fun_list2 = []
18 for i in range(5):
19     def fun2(e,iv=i):
20         return e+iv
21     fun_list2.append(fun2)
22
23 fun_list3 = [lambda e: e+i for i in range(5)]
24
25 fun_list4 = [lambda e,iv=i: e+iv for i in range(5)]
26
27 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)
29 # in Shell do
30 ## ipython -i pythonDemo.py
31 # Try these (copy text after the comment symbol and paste in the Python prompt):
32 # print([f(10) for f in fun_list1])
33 # print([f(10) for f in fun_list2])
34 # print([f(10) for f in fun_list3])
35 # print([f(10) for f in fun_list4])

```

In the first for-loop, the function *fun* 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 it is possible to add a `__doc__` string, which is the standard for documenting functions in Python, to the embedded definitions.

<sup>2</sup>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.

### 1.5.3 Generators and Coroutines

Python has generators which can be used for a form of coroutines.

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.

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

```
pythonDemo.py — (continued)
37 def myrange(start, stop, step=1):
38     """enumerates the values from start in steps of size step that are
39     less than stop.
40     """
41     assert step>0, "only positive steps implemented in myrange"
42     i = start
43     while i<stop:
44         yield i
45         i += step
46
47 print("myrange(2,30,3):",list(myrange(2,30,3)))
```

Note that 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. Note also that the built-in *range* also allows for indexing (e.g., *range(2,30,3)[2]* returns 8), which the above implementation does not. However *myrange* also works for floats, which 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.)

*Yield* can be used to generate the same sequence of values as in the example of Section 1.5.1:

```
pythonDemo.py — (continued)
49 def ga(n):
50     """generates square of even nonnegative integers less than n"""
51     for e in range(n):
52         if e%2==0:
53             yield e*e
54 a = ga(20)
```

The sequence of *next(a)*, and *list(a)* gives exactly the same results as the comprehension in Section 1.5.1.

It is straightforward to write a version of the built-in *enumerate*. Let's call it *myenumerate*:

```
pythonDemo.py — (continued)
56 def myenumerate(enum):
57     for i in range(len(enum)):
58         yield i,enum[i]
```

**Exercise 1.2** Write a version of *enumerate* where the only iteration is “for val in enum”. Hint: keep track of the index.

## 1.6 Useful Libraries

### 1.6.1 Timing Code

In order to compare algorithms, we often want to compute how long a program takes; this is called the **runtime** of the program. The most straightforward way to compute runtime is to use *time.perf\_counter()*, as in:

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

If this time is very small (say less than 0.2 second), it is probably very inaccurate, and it may be better to run your code many 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 variable *number* should be set so that the runtime 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. Usually the minimum time is the one to report, but you should be explicit and explain what you are reporting.

### 1.6.2 Plotting: Matplotlib

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

Here is a simple example that uses everything we will use.

```
pythonDemo.py — (continued)
60 import matplotlib.pyplot as plt
61
62 def myplot(min,max,step,fun1,fun2):
63     plt.ion() # make it interactive
64     plt.xlabel("The x axis")
65     plt.ylabel("The y axis")
```

```

66     plt.xscale('linear') # Makes a 'log' or 'linear' scale
67     xvalues = range(min,max,step)
68     plt.plot(xvalues,[fun1(x) for x in xvalues],
69              label="The first fun")
70     plt.plot(xvalues,[fun2(x) for x in xvalues], linestyle='--',color='k',
71              label=fun2.__doc__) # use the doc string of the function
72     plt.legend(loc="upper right") # display the legend
73
74     def slin(x):
75         """y=2x+7"""
76         return 2*x+7
77     def sqfun(x):
78         """y=(x-40)^2/10-20"""
79         return (x-40)**2/10-20
80
81     # Try the following:
82     # from pythonDemo import myplot, slin, sqfun
83     # import matplotlib.pyplot as plt
84     # myplot(0,100,1,slin,sqfun)
85     # plt.legend(loc="best")
86     # import math
87     # plt.plot([41+40*math.cos(th/10) for th in range(50)],
88              # [100+100*math.sin(th/10) for th in range(50)])
89     # plt.text(40,100,"ellipse?")
90     # plt.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

In this distribution, to keep things simple and to only use standard Python, we use a text-oriented tracing of the code. A graphical depiction of the code could override the definition of *display* (but we leave it as a project).

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:

```

_____utilities.py — AIFCA utilities_____
11 class Displayable(object):
12     max_display_level = 1 # can be overridden in subclasses

```

```

13
14     def display(self, level, *args, **nargs):
15         """print the arguments if level is less than or equal to the
16         current max_display_level.
17         level is an integer.
18         the other arguments are whatever arguments print can take.
19         """
20         if level <= self.max_display_level:
21             print(*args, **nargs) ##if error you are using Python2 not Python3

```

Note that *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 say 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
- 2 also display the values as they change
- 3 also display more details

### 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 *argmax* method returns the index of an element that has the maximum value. If there are multiple elements with the maximum value, one if the indexes to that value is returned at random. This assumes a generator of (*element, value*) pairs, as for example is generated by the built-in *enumerate*.

```

_____utilities.py — (continued)_____
23 import random
24
25 def argmax(gen):
26     """gen is a generator of (element,value) pairs, where value is a real.
27     argmax returns an element with maximal value.
28     If there are multiple elements with the max value, one is returned at random.
29     """
30     maxv = float('-Infinity')    # negative infinity

```

```

31     maxvals = []      # list of maximal elements
32     for (e,v) in gen:
33         if v>maxv:
34             maxvals,maxv = [e], v
35         elif v==maxv:
36             maxvals.append(e)
37     return random.choice(maxvals)
38
39 # Try:
40 # argmax(enumerate([1,6,3,77,3,55,23]))

```

**Exercise 1.3** Change `argmax` 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.

### 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) _____
42 def flip(prob):
43     """return true with probability prob"""
44     return random.random() < prob

```

### 1.7.4 Dictionary Union

The function `dict_union( $d1$ ,  $d2$ )` returns the union of dictionaries  $d1$  and  $d2$ . If the values for the keys conflict, the values in  $d2$  are used. This is similar to `dict( $d1$ , ** $d2$ )`, but that only works when the keys of  $d2$  are strings.

```

_____utilities.py — (continued) _____
46 def dict_union(d1,d2):
47     """returns a dictionary that contains the keys of d1 and d2.
48     The value for each key that is in d2 is the value from d2,
49     otherwise it is the value from d1.
50     This does not have side effects.
51     """
52     d = dict(d1) # copy d1
53     d.update(d2)
54     return d

```

## 1.8 Testing Code

It is important to test code early and test it often. We include a simple form of **unit tests**. The value of the current module is in `__name__` and if the module is



run at the top-level, it's value is `"__main__"`. See [https://docs.python.org/3/library/\\_\\_main\\_\\_.html](https://docs.python.org/3/library/__main__.html).

The following code tests `argmax` and `dict_union`, but only when `utilities` is loaded in the top-level. If it is loaded in a module the test code is not run.

In your code you should do more substantial testing than we do here, in particular testing the boundary cases.

```
_____utilities.py — (continued) _____  
56 def test():  
57     """Test part of utilities"""  
58     assert argmax(enumerate([1,6,55,3,55,23])) in [2,4]  
59     assert dict_union({1:4, 2:5, 3:4},{5:7, 2:9}) == {1:4, 2:9, 3:4, 5:7}  
60     print("Passed unit test in utilities")  
61  
62 if __name__ == "__main__":  
63     test()
```



## Agents and Control

This implements the controllers described in Chapter 2.

In this version the higher-levels call the lower-levels. A more sophisticated version may have them run concurrently (either as coroutines or in parallel). The higher-levels calling the lower-level works in simulated environments when there is a single agent, and 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).

### 2.1 Representing Agents and Environments

An agent observes the world, and carries out actions in the environment, it also has an internal state that it updates. The environment takes in actions of the agents, updates its internal state and returns the percepts.

In this implementation, 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 percepts and the actions are represented as variable-value dictionaries.

An agent implements the  $go(n)$  method, where  $n$  is an integer. This means that the agent should run for  $n$  time steps.

In the following code `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 —  
11 import random  
12  
13 class Agent(object):  
14     def __init__(self, env):
```

```

15     """set up the agent"""
16     self.env=env
17
18     def go(self,n):
19         """acts for n time steps"""
20         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 *Environment* is a subclass of *Displayable* so that it can use the *display* method described in Section 1.7.1.

---

agents.py — (continued)

---

```

22 from utilities import Displayable
23 class Environment(Displayable):
24     def initial_percepts(self):
25         """returns the initial percepts for the agent"""
26         raise NotImplementedError("initial_percepts") # abstract method
27
28     def do(self,action):
29         """does the action in the environment
30         returns the next percept """
31         raise NotImplementedError("do") # abstract method

```

## 2.2 Paper buying agent and environment

To run the demo, in folder "aipython", load "agents.py", using e.g., `ipython -i agents.py`, and copy and paste the commented-out commands at the bottom of that file. This requires Python 3 with matplotlib.

This is an implementation of the paper buying example.

### 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 percepts are the price and the amount of paper in stock. The action of the agent is the number to buy.

Here we assume that the prices are obtained from the *prices* list plus a random integer in range  $[0, \text{max\_price\_addon})$  plus a linear "inflation". The agent cannot access the price model; it just observes the prices and the amount in stock.

---

agents.py — (continued)

---

```

33 class TP_env(Environment):

```

```

34 prices = [234, 234, 234, 234, 255, 255, 275, 275, 211, 211, 211,
35 234, 234, 234, 234, 199, 199, 275, 275, 234, 234, 234, 234, 255,
36 255, 260, 260, 265, 265, 265, 265, 270, 270, 255, 255, 260, 260,
37 265, 265, 150, 150, 265, 265, 270, 270, 255, 255, 260, 260, 265,
38 265, 265, 265, 270, 270, 211, 211, 255, 255, 260, 260, 265, 265,
39 260, 265, 270, 270, 205, 255, 255, 260, 260, 265, 265, 265, 265,
40 270, 270]
41 max_price_addon = 20 # maximum of random value added to get price
42
43 def __init__(self):
44     """paper buying agent"""
45     self.time=0
46     self.stock=20
47     self.stock_history = [] # memory of the stock history
48     self.price_history = [] # memory of the price history
49
50 def initial_percepts(self):
51     """return initial percepts"""
52     self.stock_history.append(self.stock)
53     price = self.prices[0]+random.randrange(self.max_price_addon)
54     self.price_history.append(price)
55     return {'price': price,
56           'instock': self.stock}
57
58 def do(self, action):
59     """does action (buy) and returns percepts (price and instock)"""
60     used = pick_from_dist({6:0.1, 5:0.1, 4:0.2, 3:0.3, 2:0.2, 1:0.1})
61     bought = action['buy']
62     self.stock = self.stock+bought-used
63     self.stock_history.append(self.stock)
64     self.time += 1
65     price = (self.prices[self.time%len(self.prices)] # repeating pattern
66             +random.randrange(self.max_price_addon) # plus randomness
67             +self.time//2) # plus inflation
68     self.price_history.append(price)
69     return {'price': price,
70           'instock':self.stock}

```

The *pick\_from\_dist* method takes in a *item : probability* dictionary, and returns one of the items in proportion to its probability.

---

agents.py — (continued)

---

```

72 def pick_from_dist(item_prob_dist):
73     """ returns a value from a distribution.
74     item_prob_dist is an item:probability dictionary, where the
75     probabilities sum to 1.
76     returns an item chosen in proportion to its probability
77     """
78     ranreal = random.random()
79     for (it,prob) in item_prob_dist.items():
80         if ranreal < prob:

```

```

81         return it
82     else:
83         ranreal -= prob
84     raise RuntimeError(str(item_prob_dist)+" is not a probability distribution")

```

### 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.

```

agents.py — (continued)
86 class TP_agent(Agent):
87     def __init__(self, env):
88         self.env = env
89         self.spent = 0
90         percepts = env.initial_percepts()
91         self.ave = self.last_price = percepts['price']
92         self.instock = percepts['instock']
93
94     def go(self, n):
95         """go for n time steps
96         """
97         for i in range(n):
98             if self.last_price < 0.9*self.ave and self.instock < 60:
99                 tobuy = 48
100             elif self.instock < 12:
101                 tobuy = 12
102             else:
103                 tobuy = 0
104             self.spent += tobuy*self.last_price
105             percepts = env.do({'buy': tobuy})
106             self.last_price = percepts['price']
107             self.ave = self.ave+(self.last_price-self.ave)*0.05
108             self.instock = percepts['instock']

```

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

```

agents.py — (continued)
110 env = TP_env()
111 ag = TP_agent(env)
112 #ag.go(90)
113 #ag.spent/env.time ## average spent per time period

```

### 2.2.3 Plotting

The following plots the price and number in stock history:

```

agents.py — (continued)
115 import matplotlib.pyplot as plt
116
117 class Plot_prices(object):
118     """Set up the plot for history of price and number in stock"""
119     def __init__(self, ag, env):
120         self.ag = ag
121         self.env = env
122         plt.ion()
123         plt.xlabel("Time")
124         plt.ylabel("Number in stock.                                Price.")
125
126     def plot_run(self):
127         """plot history of price and instock"""
128         num = len(env.stock_history)
129         plt.plot(range(num), env.stock_history, label="In stock")
130         plt.plot(range(num), env.price_history, label="Price")
131         #plt.legend(loc="upper left")
132         plt.draw()
133
134 # pl = Plot_prices(ag, env)
135 # ag.go(90); pl.plot_run()

```

## 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. This requires Python 3 with matplotlib.

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.

We arbitrarily divide the environment and the body, so that the environment just defines the walls, and the body includes everything to do with the agent. Note that the named locations are part of the (top-level of the) agent, not part of the environment, although they could have been.

### 2.3.1 Environment

The environment defines the walls.

```

agentEnv.py — Agent environment
11 import math
12 from agents import Environment
13
14 class Rob_env(Environment):

```

```

15     def __init__(self, walls = {}):
16         """walls is a set of line segments
17             where each line segment is of the form ((x0,y0),(x1,y1))
18         """
19         self.walls = walls

```

### 2.3.2 Body

The body defines everything about the agent body.

```

agentEnv.py — (continued)
21 import math
22 from agents import Environment
23 import matplotlib.pyplot as plt
24 import time
25
26 class Rob_body(Environment):
27     def __init__(self, env, init_pos=(0,0,90)):
28         """ env is the current environment
29             init_pos is a triple of (x-position, y-position, direction)
30             direction is in degrees; 0 is to right, 90 is straight-up, etc
31         """
32         self.env = env
33         self.rob_x, self.rob_y, self.rob_dir = init_pos
34         self.turning_angle = 18 # degrees that a left makes
35         self.whisker_length = 6 # length of the whisker
36         self.whisker_angle = 30 # angle of whisker relative to robot
37         self.crashed = False
38         # The following control how it is plotted
39         self.plotting = True # whether the trace is being plotted
40         self.sleep_time = 0.05 # time between actions (for real-time plotting)
41         # The following are data structures maintained:
42         self.history = [(self.rob_x, self.rob_y)] # history of (x,y) positions
43         self.wall_history = [] # history of hitting the wall
44
45     def percepts(self):
46         return {'rob_x_pos':self.rob_x, 'rob_y_pos':self.rob_y,
47             'rob_dir':self.rob_dir, 'whisker':self.whisker() , 'crashed':self.crashed}
48     initial_percepts = percepts # use percept function for initial percepts too
49
50     def do(self, action):
51         """ action is {'steer':direction}
52             direction is 'left', 'right' or 'straight'
53         """
54         if self.crashed:
55             return self.percepts()
56         direction = action['steer']
57         compass_deriv = {'left':1, 'straight':0, 'right':-1}[direction]*self.turning_angle
58         self.rob_dir = (self.rob_dir + compass_deriv + 360)%360 # make in range [0,360)
59         rob_x_new = self.rob_x + math.cos(self.rob_dir*math.pi/180)

```



```

60     rob_y_new = self.rob_y + math.sin(self.rob_dir*math.pi/180)
61     path = ((self.rob_x,self.rob_y),(rob_x_new,rob_y_new))
62     if any(line_segments_intersect(path,wall) for wall in self.env.walls):
63         self.crashed = True
64         if self.plotting:
65             plt.plot([self.rob_x],[self.rob_y],"r*",markersize=20.0)
66             plt.draw()
67     self.rob_x, self.rob_y = rob_x_new, rob_y_new
68     self.history.append((self.rob_x, self.rob_y))
69     if self.plotting and not self.crashed:
70         plt.plot([self.rob_x],[self.rob_y],"go")
71         plt.draw()
72         plt.pause(self.sleep_time)
73     return self.percepts()

```

This detects if the whisker and the wall intersect. It's value is returned as a percept.

agentEnv.py — (continued)

```

75     def whisker(self):
76         """returns true whenever the whisker sensor intersects with a wall
77         """
78         whisk_ang_world = (self.rob_dir-self.whisker_angle)*math.pi/180
79         # angle in radians in world coordinates
80         wx = self.rob_x + self.whisker_length * math.cos(whisk_ang_world)
81         wy = self.rob_y + self.whisker_length * math.sin(whisk_ang_world)
82         whisker_line = ((self.rob_x,self.rob_y),(wx,wy))
83         hit = any(line_segments_intersect(whisker_line,wall)
84                 for wall in self.env.walls)
85         if hit:
86             self.wall_history.append((self.rob_x, self.rob_y))
87             if self.plotting:
88                 plt.plot([self.rob_x],[self.rob_y],"ro")
89                 plt.draw()
90         return hit
91
92     def line_segments_intersect(linea,lineb):
93         """returns true if the line segments, linea and lineb intersect.
94         A line segment is represented as a pair of points.
95         A point is represented as a (x,y) pair.
96         """
97         ((x0a,y0a),(x1a,y1a)) = linea
98         ((x0b,y0b),(x1b,y1b)) = lineb
99         da, db = x1a-x0a, x1b-x0b
100        ea, eb = y1a-y0a, y1b-y0b
101        denom = db*ea-eb*da
102        if denom==0: # line segments are parallel
103            return False
104        cb = (da*(y0b-y0a)-ea*(x0b-x0a))/denom # position along line b
105        if cb<0 or cb>1:
106            return False

```

```

107     ca = (db*(y0b-y0a)-eb*(x0b-x0a))/denom # position along line a
108     return 0<=ca<=1
109
110 # Test cases:
111 # assert line_segments_intersect(((0,0),(1,1)),((1,0),(0,1)))
112 # assert not line_segments_intersect(((0,0),(1,1)),((1,0),(0.6,0.4)))
113 # assert line_segments_intersect(((0,0),(1,1)),((1,0),(0.4,0.6)))

```

### 2.3.3 Middle Layer

The middle layer acts like both a controller (for the environment layer) and an environment for the upper layer. It has to tell the environment how to steer. Thus it calls *env.do(·)*. It also is told the position to go to and the timeout. Thus it also has to implement *do(·)*.

```

agentMiddle.py — Middle Layer
11 from agents import Environment
12 import math
13
14 class Rob_middle_layer(Environment):
15     def __init__(self,env):
16         self.env=env
17         self.percepts = env.initial_percepts()
18         self.straight_angle = 11 # angle that is close enough to straight ahead
19         self.close_threshold = 2 # distance that is close enough to arrived
20         self.close_threshold_squared = self.close_threshold**2 # just compute it once
21
22     def initial_percepts(self):
23         return {}
24
25     def do(self, action):
26         """action is {'go_to':target_pos,'timeout':timeout}
27         target_pos is (x,y) pair
28         timeout is the number of steps to try
29         returns {'arrived':True} when arrived is true
30         or {'arrived':False} if it reached the timeout
31         """
32         if 'timeout' in action:
33             remaining = action['timeout']
34         else:
35             remaining = -1 # will never reach 0
36         target_pos = action['go_to']
37         arrived = self.close_enough(target_pos)
38         while not arrived and remaining != 0:
39             self.percepts = self.env.do({"steer":self.steer(target_pos)})
40             remaining -= 1
41             arrived = self.close_enough(target_pos)
42         return {'arrived':arrived}

```

This 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)
44 def steer(self, target_pos):
45     if self.percepts['whisker']:
46         self.display(3, 'whisker on', self.percepts)
47         return "left"
48     else:
49         gx, gy = target_pos
50         rx, ry = self.percepts['rob_x_pos'], self.percepts['rob_y_pos']
51         goal_dir = math.acos((gx-rx)/math.sqrt((gx-rx)*(gx-rx)
52                                     +(gy-ry)*(gy-ry)))*180/math.pi
53         if ry>gy:
54             goal_dir = -goal_dir
55         goal_from_rob = (goal_dir - self.percepts['rob_dir']+540)%360-180
56         assert -180 < goal_from_rob <= 180
57         if goal_from_rob > self.straight_angle:
58             return "left"
59         elif goal_from_rob < -self.straight_angle:
60             return "right"
61         else:
62             return "straight"
63
64 def close_enough(self, target_pos):
65     gx, gy = target_pos
66     rx, ry = self.percepts['rob_x_pos'], self.percepts['rob_y_pos']
67     return (gx-rx)**2 + (gy-ry)**2 <= self.close_threshold_squared

```

### 2.3.4 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 agentMiddle import Rob_middle_layer
12 from agents import Environment
13
14 class Rob_top_layer(Environment):
15     def __init__(self, middle, timeout=200, locations = {'mail':(-5,10),
16                                                         'o103':(50,10), 'o109':(100,10), 'storage':(101,51)} ):
17         """middle is the middle layer
18         timeout is the number of steps the middle layer goes before giving up
19         locations is a loc:pos dictionary
20         where loc is a named location, and pos is an (x,y) position.
21         """
22         self.middle = middle
23         self.timeout = timeout # number of steps before the middle layer should give up
24         self.locations = locations
25

```

```

26     def do(self, plan):
27         """carry out actions.
28         actions is of the form {'visit':list_of_locations}
29         It visits the locations in turn.
30         """
31         to_do = plan['visit']
32         for loc in to_do:
33             position = self.locations[loc]
34             arrived = self.middle.do({'go_to':position, 'timeout':self.timeout})
35             self.display(1, "Arrived at", loc, arrived)

```

### 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()`).

```

agentTop.py — (continued)
37 import matplotlib.pyplot as plt
38
39 class Plot_env(object):
40     def __init__(self, body, top):
41         """sets up the plot
42         """
43         self.body = body
44         plt.ion()
45         plt.clf()
46         plt.axes().set_aspect('equal')
47         for wall in body.env.walls:
48             ((x0,y0),(x1,y1)) = wall
49             plt.plot([x0,x1],[y0,y1], "-k", linewidth=3)
50         for loc in top.locations:
51             (x,y) = top.locations[loc]
52             plt.plot([x],[y], "k<")
53             plt.text(x+1.0,y+0.5,loc) # print the label above and to the right
54         plt.plot([body.rob_x],[body.rob_y], "go")
55         plt.draw()
56
57     def plot_run(self):
58         """plots the history after the agent has finished.
59         This is typically only used if body.plotting==False
60         """
61         xs,ys = zip(*self.body.history)
62         plt.plot(xs,ys, "go")
63         wxs,wys = zip(*self.body.wall_history)
64         plt.plot(wxs,wys, "ro")
65         #plt.draw()

```

The following code plots the agent as it acts in the world:

```

agentTop.py — (continued)
67 from agentEnv import Rob_body, Rob_env
68
69 env = Rob_env({((20,0),(30,20)), ((70,-5),(70,25))})
70 body = Rob_body(env)
71 middle = Rob_middle_layer(body)
72 top = Rob_top_layer(middle)
73
74 # try:
75 # pl=Plot_env(body,top)
76 # top.do({'visit':['o109','storage','o109','o103']})
77 # You can directly control the middle layer:
78 # middle.do({'go_to':(30,-10), 'timeout':200})
79 # Can you make it crash?

```

**Exercise 2.1** The following code implements a robot trap. Write a controller that can escape the “trap” and get to the goal. See textbook for hints.

```

agentTop.py — (continued)
81 # Robot Trap for which the current controller cannot escape:
82 trap_env = Rob_env({((10,-21),(10,0)), ((10,10),(10,31)), ((30,-10),(30,0)),
83                     ((30,10),(30,20)), ((50,-21),(50,31)), ((10,-21),(50,-21)),
84                     ((10,0),(30,0)), ((10,10),(30,10)), ((10,31),(50,31))})
85 trap_body = Rob_body(trap_env,init_pos=(-1,0,90))
86 trap_middle = Rob_middle_layer(trap_body)
87 trap_top = Rob_top_layer(trap_middle,locations={'goal':(71,0)})
88
89 # Robot trap exercise:
90 # pl=Plot_env(trap_body,trap_top)
91 # trap_top.do({'visit':['goal']})

```



## Searching for Solutions

### 3.1 Representing Search Problems

A search problem consists of:

- a set of start nodes
- 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 class Search_problem(object):
12     """A search problem consists of:
13     * a set of start nodes
14     * a neighbors function that gives the neighbors of a node
15     * a specification of a goal
16     * a (optional) heuristic function.
```

```

17     The methods must be overridden to define a search problem."""
18
19     def start_nodes(self):
20         """returns list of start nodes"""
21         raise NotImplementedError("start_nodes") # abstract method
22
23     def is_goal(self,node):
24         """is True if node is a goal"""
25         raise NotImplementedError("is_goal") # abstract method
26
27     def neighbors(self,node):
28         """returns a list of the arcs for the neighbors of node"""
29         raise NotImplementedError("neighbors") # abstract method
30
31     def heuristic(self,n):
32         """Gives the heuristic value of node n.
33         Returns 0 if not overridden."""
34         return 0

```

The neighbors is a list of arcs. A (directed) arc consists of a *from\_node* node and a *to\_node* node. The arc is the pair  $\langle from\_node, to\_node \rangle$ , but can also contain a non-negative *cost* (which defaults to 1) and can be labeled with an *action*.

searchProblem.py — (continued)

```

36 class Arc(object):
37     """An arc has a from_node and a to_node node and a (non-negative) cost"""
38     def __init__(self, from_node, to_node, cost=1, action=None):
39         assert cost >= 0, ("Cost cannot be negative for"+
40                             str(from_node)+"-->"+str(to_node)+"", cost: "+str(cost))
41         self.from_node = from_node
42         self.to_node = to_node
43         self.action = action
44         self.cost=cost
45
46     def __repr__(self):
47         """string representation of an arc"""
48         if self.action:
49             return str(self.from_node)+" --"+str(self.action)+"--> "+str(self.to_node)
50         else:
51             return str(self.from_node)+" --> "+str(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 list or set of start nodes
- a list or set of goal nodes
- (optionally) a dictionary that maps a node to a heuristic value for that node

To define a search problem, we need to define the start nodes, the goal predicate, the neighbors function and the heuristic function.

```

53 class Search_problem_from_explicit_graph(Search_problem):
54     """A search problem consists of:
55     * a list or set of nodes
56     * a list or set of arcs
57     * a list or set of start nodes
58     * a list or set of goal nodes
59     * a dictionary that maps each node into its heuristic value.
60     """
61
62     def __init__(self, nodes, arcs, starts=[], goals=set(), hmap={}):
63         self.neighs = {}
64         self.nodes = nodes
65         for node in nodes:
66             self.neighs[node]=[]
67         self.arcs = arcs
68         for arc in arcs:
69             self.neighs[arc.from_node].append(arc)
70         self.starts = starts
71         self.goals = goals
72         self.hmap = hmap
73
74     def start_nodes(self):
75         """returns list of start nodes"""
76         return self.starts
77
78     def is_goal(self,node):
79         """is True if node is a goal"""
80         return node in self.goals
81
82     def neighbors(self,node):
83         """returns the neighbors of node"""
84         return self.neighs[node]
85
86     def heuristic(self,node):
87         """Gives the heuristic value of node n.
88         Returns 0 if not overridden in the hmap."""
89         if node in self.hmap:
90             return self.hmap[node]
91         else:
92             return 0

```

```

93
94     def __repr__(self):
95         """returns a string representation of the search problem"""
96         res=""
97         for arc in self.arcs:
98             res += str(arc)+" "
99         return res

```

The following is used for the depth-first search implementation below.

```

searchProblem.py — (continued)
101     def neighbor_nodes(self,node):
102         """returns an iterator over the neighbors of node"""
103         return (path.to_node for path in self.neighs[node])

```

### 3.1.2 Paths

A searcher will return a path from a 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
- a path, *initial* and an arc, where the *from\_node* of the arc is the node at the end of *initial*.

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.

```

searchProblem.py — (continued)
105 class Path(object):
106     """A path is either a node or a path followed by an arc"""
107
108     def __init__(self,initial,arc=None):
109         """initial is either a node (in which case arc is None) or
110         a path (in which case arc is an object of type Arc)"""
111         self.initial = initial
112         self.arc=arc
113         if arc is None:
114             self.cost=0
115         else:
116             self.cost = initial.cost+arc.cost
117
118     def end(self):
119         """returns the node at the end of the path"""
120         if self.arc is None:

```

```

121         return self.initial
122     else:
123         return self.arc.to_node
124
125     def nodes(self):
126         """enumerates the nodes for the path.
127         This starts at the end and enumerates nodes in the path backwards."""
128         current = self
129         while current.arc is not None:
130             yield current.arc.to_node
131             current = current.initial
132         yield current.initial
133
134     def initial_nodes(self):
135         """enumerates the nodes for the path before the end node.
136         This starts at the end and enumerates nodes in the path backwards."""
137         if self.arc is not None:
138             yield from self.initial.nodes()
139
140     def __repr__(self):
141         """returns a string representation of a path"""
142         if self.arc is None:
143             return str(self.initial)
144         elif self.arc.action:
145             return (str(self.initial)+"\n --"+str(self.arc.action)
146                     +"--> "+str(self.arc.to_node))
147         else:
148             return str(self.initial)+" --> "+str(self.arc.to_node)

```

### 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. Note that this example is used for the unit tests, so the test (in `searchAStar`) will need to be changed if this is changed.

```

searchProblem.py — (continued)
150 problem1 = Search_problem_from_explicit_graph(
151     {'a','b','c','d','g'},
152     [Arc('a','b',1), Arc('a','c',3), Arc('b','c',1), Arc('b','d',3),
153       Arc('c','d',1), Arc('c','g',3), Arc('d','g',1)],
154     starts = ['a'],
155     goals = {'g'})

```

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

```

searchProblem.py — (continued)
157 problem2 = Search_problem_from_explicit_graph(
158     {'a','b','c','d','e','g','h','j'},
159     [Arc('a','b',1), Arc('b','c',3), Arc('b','d',1), Arc('d','e',3),

```

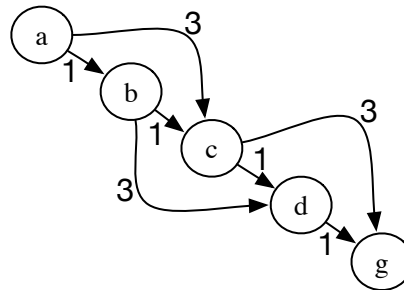


Figure 3.1: problem1

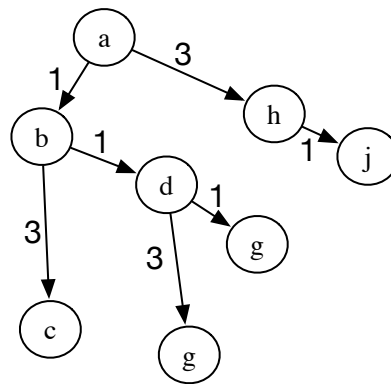


Figure 3.2: problem2

```

160     Arc('d','g',1), Arc('a','h',3), Arc('h','j',1)],
161     starts = ['a'],
162     goals = {'g'})

```

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

searchProblem.py — (continued)

```

164 problem3 = Search_problem_from_explicit_graph(
165     {'a','b','c','d','e','g','h','j'},
166     [],
167     starts = ['a','b','g','h'],
168     goals = {'k','g'})

```

The `acyclic_delivery_problem` is the delivery problem described in Example 3.4 and shown in Figure 3.2 of the textbook.

searchProblem.py — (continued)

```

170 acyclic_delivery_problem = Search_problem_from_explicit_graph(
171     {'mail','ts','o103','o109','o111','b1','b2','b3','b4','c1','c2','c3',
172      'o125','o123','o119','r123','storage'},
173     [Arc('ts','mail',6),
174        Arc('o103','ts',8),
175        Arc('o103','b3',4),
176        Arc('o103','o109',12),
177        Arc('o109','o119',16),
178        Arc('o109','o111',4),
179        Arc('b1','c2',3),
180        Arc('b1','b2',6),
181        Arc('b2','b4',3),
182        Arc('b3','b1',4),
183        Arc('b3','b4',7),
184        Arc('b4','o109',7),
185        Arc('c1','c3',8),
186        Arc('c2','c3',6),
187        Arc('c2','c1',4),
188        Arc('o123','o125',4),
189        Arc('o123','r123',4),
190        Arc('o119','o123',9),
191        Arc('o119','storage',7)],
192     starts = ['o103'],
193     goals = {'r123'},
194     hmap = {
195         'mail' : 26,
196         'ts' : 23,
197         'o103' : 21,
198         'o109' : 24,
199         'o111' : 27,
200         'o119' : 11,
201         'o123' : 4,
202         'o125' : 6,
203         'r123' : 0,
204         'b1' : 13,
205         'b2' : 15,
206         'b3' : 17,
207         'b4' : 18,
208         'c1' : 6,
209         'c2' : 10,
210         'c3' : 12,
211         'storage' : 12
212     }
213 )

```

The cyclic\_delivery\_problem is the delivery problem described in Example 3.8 and shown in Figure 3.6. This is the same as acyclic\_delivery\_problem, but almost every arc also has its inverse.

---

searchProblem.py — (continued)

---

```

215 cyclic_delivery_problem = Search_problem_from_explicit_graph(

```

```

216 {'mail','ts','o103','o109','o111','b1','b2','b3','b4','c1','c2','c3',
217     'o125','o123','o119','r123','storage'},
218 [ Arc('ts','mail',6), Arc('mail','ts',6),
219   Arc('o103','ts',8), Arc('ts','o103',8),
220   Arc('o103','b3',4),
221   Arc('o103','o109',12), Arc('o109','o103',12),
222   Arc('o109','o119',16), Arc('o119','o109',16),
223   Arc('o109','o111',4), Arc('o111','o109',4),
224   Arc('b1','c2',3),
225   Arc('b1','b2',6), Arc('b2','b1',6),
226   Arc('b2','b4',3), Arc('b4','b2',3),
227   Arc('b3','b1',4), Arc('b1','b3',4),
228   Arc('b3','b4',7), Arc('b4','b3',7),
229   Arc('b4','o109',7),
230   Arc('c1','c3',8), Arc('c3','c1',8),
231   Arc('c2','c3',6), Arc('c3','c2',6),
232   Arc('c2','c1',4), Arc('c1','c2',4),
233   Arc('o123','o125',4), Arc('o125','o123',4),
234   Arc('o123','r123',4), Arc('r123','o123',4),
235   Arc('o119','o123',9), Arc('o123','o119',9),
236   Arc('o119','storage',7), Arc('storage','o119',7)],
237 starts = ['o103'],
238 goals = {'r123'},
239 hmap = {
240     'mail' : 26,
241     'ts' : 23,
242     'o103' : 21,
243     'o109' : 24,
244     'o111' : 27,
245     'o119' : 11,
246     'o123' : 4,
247     'o125' : 6,
248     'r123' : 0,
249     'b1' : 13,
250     'b2' : 15,
251     'b3' : 17,
252     'b4' : 18,
253     'c1' : 6,
254     'c2' : 10,
255     'c3' : 12,
256     'storage' : 12
257 }
258 )

```

## 3.2 $A^*$ and Related Searchers

To run the  $A^*$  demo, in folder “aipython”, load “searchAStar.py”, using e.g., `ipython -i searchAStar.py`, and copy and paste the example queries at the bottom of that file. This requires Python 3.

### 3.2.1 Frontier

In many of the search algorithms, such as  $A^*$  and other best-first searchers, the frontier is implemented as a priority queue. Here we use the Python’s built-in priority queue implementations, `heapq`.

Following the lead of the Python documentation, <http://docs.python.org/3.3/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 when the first elements are the same, 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 a unique index, it is useful for statistics, particularly in conjunction with the current size of the frontier.

```

searchAStar.py —  $A^*$  and related searchers
11 import heapq          # part of the Python standard library
12 from searchProblem import Path
13
14 class Frontier(object):
15     """A frontier consists of a priority queue (heap), frontierpq, of
16        (value, index, path) triples, where
17        * value is the value we want to minimize (e.g., path cost + h).
18        * index is a unique index for each element
19        * path is the path on the queue
20        Note that the priority queue always returns the smallest element.
21     """
22
23     def __init__(self):
24         """constructs the frontier, initially an empty priority queue
25         """
26         self.frontier_index = 0 # the number of items ever added to the frontier
27         self.frontierpq = [] # the frontier priority queue
28
29     def empty(self):
30         """is True if the priority queue is empty"""
31         return self.frontierpq == []
32
33     def add(self, path, value):

```

```

34         """add a path to the priority queue
35         value is the value to be minimized"""
36         self.frontier_index += 1 # get a new unique index
37         heapq.heappush(self.frontierpq, (value, -self.frontier_index, path))
38
39     def pop(self):
40         """returns and removes the path of the frontier with minimum value.
41         Note that [2] extracts the path from the triple on the queue.
42         """
43         return heapq.heappop(self.frontierpq)[2]

```

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

```

searchAStar.py — (continued)
45     def count(self, val):
46         """returns the number of elements of the frontier with value=val"""
47         return sum(1 for e in self.frontierpq if e[0]==val)
48
49     def __repr__(self):
50         """string representation of the frontier"""
51         return str([(n,c,str(p)) for (n,c,p) in self.frontierpq])

```

### 3.2.2 Searcher

A *Searcher* for a problem can be asked repeatedly for the next path. To solve a problem, we can 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.

```

searchAStar.py — (continued)
53 from utilities import Displayable
54
55 class Searcher(Displayable):
56     """returns a searcher for a problem.
57     Paths can be found by repeatedly calling search().
58     """
59
60     def __init__(self, problem, method='astar'):
61         """creates a searcher from a problem
62         method is 'astar' or 'best-first' or 'lowest-cost-first'
63         """
64         self.problem = problem
65         self.method = method # not used
66         self.frontier = Frontier()
67         self.num_expanded = 0
68         for node in problem.start_nodes():
69             self.add_to_frontier(Path(node))
70         self.display(3, "Frontier:", self.frontier)
71
72     def add_to_frontier(self, path):

```



```

73         """add path to the frontier with the appropriate cost"""
74         if self.method=='astar':
75             value = path.cost+self.problem.heuristic(path.end())
76         else:
77             assert False, "unknown method "+str(self.method)
78         self.frontier.add(path, value)
79
80     def search(self):
81         """returns (next) path from an element of problem's start nodes
82         to a goal node.
83         Returns None if no path exists.
84         """
85         while not self.frontier.empty():
86             path = self.frontier.pop()
87             self.display(2, "Expanding:",path,"(cost:",path.cost,")")
88             self.num_expanded += 1
89             if self.problem.is_goal(path.end()): # solution found
90                 self.display(1, self.num_expanded, "paths have been expanded and",
91                             len(self.frontier.frontierpq), "nodes remain in the frontier")
92                 self.solution = path # store the solution found
93                 return path
94             else:
95                 neighs = self.problem.neighbors(path.end())
96                 for arc in neighs:
97                     self.add_to_frontier(Path(path,arc))
98                     self.display(3,"Frontier:",self.frontier)
99         self.display(1,"No (more) solutions. Total of",
100                    self.frontier.frontier_index,"paths expanded.")

```

Testing:

```

searchAStar.py — (continued)
102 import searchProblem
103 def test(SearchClass):
104     print("Testing problem 1:")
105     schr1 = SearchClass(searchProblem.problem1)
106     path1 = schr1.search()
107     print("Path found: ",path1)
108     assert list(path1.nodes()) == ['g','d','c','b','a'], "Shortest path not found in problem1"
109     print("Passed unit test")
110
111 if __name__ == "__main__":
112     test(Searcher)
113
114 # example queries:
115 # searcher1 = Searcher(searchProblem.acyclic_delivery_problem)
116 # searcher1.search() # find first path
117 # searcher1.search() # find next path
118 # searcher2 = Searcher(searchProblem.cyclic_delivery_problem)
119 # searcher2.search() # find first path
120 # searcher2.search() # find next path

```

### 3.2.3 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
11 from searchAStar import Searcher
12 from searchProblem import Path
13
14 class SearcherMPP(Searcher):
15     """returns a searcher for a problem.
16     Paths can be found by repeatedly calling search().
17     """
18     def __init__(self, problem, method='astar'):
19         Searcher.__init__(self, problem, method)
20         self.expanded = set()
21
22     def search(self):
23         """returns next path from an element of problem's start nodes
24         to a goal node.
25         Returns None if no path exists.
26         """
27         while not self.frontier.empty():
28             path = self.frontier.pop()
29             if path.end() not in self.expanded:
30                 self.display(2, "Expanding: ", path, "(cost:", path.cost, ")")
31                 self.expanded.add(path.end())
32                 self.num_expanded += 1
33                 if self.problem.is_goal(path.end()):
34                     self.display(1, self.num_expanded, "nodes have been expanded and",
35                                 len(self.frontier.frontierpq), "nodes remain in the frontier")
36                     self.solution = path # store the solution found
37                     return path
38             else:
39                 neighs = self.problem.neighbors(path.end())
40                 for arc in neighs:
41                     self.add_to_frontier(Path(path, arc))
42                 self.display(3, "Frontier:", self.frontier)
43
44 from searchAStar import test
45 if __name__ == "__main__":
46     test(SearcherMPP)
47
48 import searchProblem
49 # searcherMPPcdp = SearcherMPP(searchProblem.cyclic_delivery_problem)
50 # print(searcherMPPcdp.search()) # find first path

```

**Exercise 3.1** 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`.) Compare no pruning, multiple path pruning and cycle pruning for the cyclic delivery problem. Which works better in terms of number of nodes expanded, computational time or space?

**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 nodes 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. Change the code so it implements the iterator interface. What does this enable us to do?

**Exercise 3.4** In the `add` method in `Frontier` what does the `"-"` in front of `frontier_index` do? When there are multiple paths with the same value, which search method does this act like? What happens if the `"-"` is removed? When there are multiple paths with the same value, which search method does this act like? Does it work better with or without the `"-"`? What evidence did you base your conclusion on?

### 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 an 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
12 from utilities import Displayable
13
14 class DF_branch_and_bound(Displayable):
15     """returns a branch and bound searcher for a problem.
16     An optimal path with cost less than bound can be found by calling search()
17     """
18     def __init__(self, problem, bound=float("inf")):
19         """creates a searcher than can be used with search() to find an optimal path.
20         bound gives the initial bound. By default this is infinite - meaning there
21         is no initial pruning due to depth bound
22         """
23         self.problem = problem

```

```

24         self.best_path = None
25         self.bound = bound
26
27     def search(self):
28         """returns an optimal solution to a problem with cost less than bound.
29         returns None if there is no solution with cost less than bound."""
30         self.number_expanded=0
31         frontier = [Path(node) for node in reversed(self.problem.start_nodes())]
32         while frontier:
33             path = frontier.pop()
34             if path.cost+self.problem.heuristic(path.end()) < self.bound:
35                 self.display(3,"Expanding:",path,"cost:",path.cost)
36                 self.number_expanded += 1
37                 if self.problem.is_goal(path.end()):
38                     self.best_path = path
39                     self.bound = path.cost
40                     self.display(2,"New best path:",path," cost:",path.cost)
41                     frontier += reversed([Path(path, arc)
42                                         for arc in self.problem.neighbors(path.end())])
43         self.display(1,"Number of nodes expanded:",self.number_expanded)
44         self.solution_path = self.best_path
45         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.

Here is a unit test and some queries:

```

_____searchBranchAndBound.py — (continued) _____
47 from searchAStar import test
48 if __name__ == "__main__":
49     test(DF_branch_and_bound)
50
51 # Example queries:
52 # import searchProblem
53 # searcherb1 = DF_branch_and_bound(searchProblem.acyclic_delivery_problem)
54 # print(searcherb1.search()) # find optimal path
55 # searcherb2 = DF_branch_and_bound(searchProblem.cyclic_delivery_problem, bound=100)
56 # print(searcherb2.search()) # find optimal path

```

**Exercise 3.5** 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 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
11 from searchAStar import Searcher
12 from searchBranchAndBound import DF_branch_and_bound
13 from searchMPP import SearcherMPP
14
15 DF_branch_and_bound.max_display_level = 1
16 Searcher.max_display_level = 1
17
18 def run(problem,name):
19     print("\n\n*****",name)
20     print("\nA*:")
21     tsearcher = Searcher(problem)
22     print("Path found:",tsearcher.search()," cost=",tsearcher.solution.cost)
23     print("there are",tsearcher.frontier.count(tsearcher.solution.cost),
24           "elements remaining on the queue with f-value=",tsearcher.solution.cost)
25
26     print("\nA* with MPP:"),
27     msearcher = SearcherMPP(problem)
28     print("Path found:",msearcher.search()," cost=",msearcher.solution.cost)
29     print("there are",msearcher.frontier.count(msearcher.solution.cost),
30           "elements remaining on the queue with f-value=",msearcher.solution.cost)
31
32     print("\nBranch and bound (with too-good initial bound):")
33     tbb = DF_branch_and_bound(problem,tsearcher.solution.cost+0.1) # cheating!!!!
34     print("Path found:",tbb.search()," cost=",tbb.solution.cost)
35
36 import searchProblem
37 from searchTest import run
38 if __name__ == "__main__":
39     run(searchProblem.problem1,"Problem 1")
40     # run(searchProblem.acyclic_delivery_problem,"Acyclic Delivery")
41     # run(searchProblem.cyclic_delivery_problem,"Cyclic Delivery")
42 # also test some graphs with cycles, and some with multiple least-cost paths

```

**Exercise 3.6** Implement a branch-and-bound search uses recursion. Hint: you don't need an explicit frontier, but can do a recursive call for the children.

## 3.4 Depth-first Search

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

Some of the algorithms in later chapters do not care about the path, but just need to find a goal node. In such cases, the searcher can be simplified. Here we give an implementation of depth-bounded depth-first search that uses a list of generators rather than a stack of nodes. Instead of going through the neighbors, and putting them on the stack, only to take them off again, this code

only generates the neighbours as needed, when they are ready to be expanded. To do this, it keeps a stack of generators of neighbouring nodes.

As it only needs the nodes (and not the arcs or the path) it uses a different interface; the problem must implement *neighbor\_nodes(node)* which returns a generator of nodes. The next neighbor can be found by calling *next()* on the generator; and the exception *StopIteration* is raised if there are no more elements in the generator. In this case, it needs to backtrack.

This does a depth-bounded search, where the bound is specified as part of the problem or in the search. If it fails due to the depth-bound, *hit\_depth\_bound* will be True.

```

_____searchDepthFirst.py — Depth-first Search for finding goals using an explicit stack of iterators_____
11 from searchProblem import Path
12 from utilities import Displayable
13
14 class Depth_first_search(Displayable):
15     """returns a depth-first searcher for a problem.
16
17     This uses a list of iterators of nodes. "top" is the top-level iterator.
18     The frontier contains iterators that may be needed to solve the problem.
19     """
20     def __init__(self, problem, bound=100000):
21         self.problem = problem
22         self.bound = bound # default bound unless overridden in search
23         self.top = iter(self.problem.start_nodes())
24         self.frontier = [] # list of iterators that generate all unexplored paths
25         self.number_expanded = 0 # number of nodes expanded
26         self.hit_depth_bound = False # true when some paths hit the depth-bound
27
28     def search(self, bound = None):
29         """finds a goal with number of arcs less than bound if there is one.
30         returns None if there is no path to a goal"""
31         if bound is None: bound = self.bound
32         while True:
33             try:
34                 node = next(self.top) #current path
35                 self.number_expanded += 1
36                 if self.problem.is_goal(node):
37                     self.display(1,"DFS found goal",node,"There were",
38                               self.number_expanded,"nodes expanded")
39                     return node
40                 elif len(self.frontier) < bound:
41                     self.frontier.append(self.top)
42                     self.display(2,"DFS expanding node",node)
43                     self.top = self.problem.neighbor_nodes(node)
44                 else:
45                     self.hit_depth_bound = True
46                     self.display(2,"DFS hit depth bound at node",node)
47             except StopIteration: #top has no more elements
48                 self.display(2,"popping off frontier")

```

```
49         if self.frontier:
50             self.top = self.frontier.pop()
51         else:
52             self.display(1,"No path found. There were",
53                         self.number_expanded,"nodes expanded")
54             return None
55
56 # example queries:
57 # from searchDepthFirst import Depth_first_search
58 # import searchProblem
59 # searcher1 = Depth_first_search(searchProblem.acyclic_delivery_problem)
60 # print(searcher1.search())    # find next path
61 # searcher2 = Depth_first_search(searchProblem.problem2)
62 # print(searcher2.search())    # find next path
63 # searcher3 = Depth_first_search(searchProblem.cyclic_delivery_problem)
64 # s3=searcher3.search()    # find next path - will go forever(?)
```

**Exercise 3.7** Write an iterative deepening searcher that will find paths in the same order as breadth-first search. Make sure it only returns each path once.





## Features and Constraints

### 4.1 Constraint Satisfaction Problems

#### 4.1.1 Constraints

A constraint consists of a tuple (or list) of variables and a condition. The tuple of variables is called the **scope**. Variables are strings. The **condition** is a Boolean function that takes the same number of arguments as there are variables in the scope. We assume that the condition has a `__name__` property that gives a printable name of the function; built-in functions and functions that are defined using *def* have such a property; for other functions you may need to define this property.

```
_____cspProblem.py — Representations of a Constraint Satisfaction Problem _____
11 from utilities import Displayable, dict_union
12
13 class Constraint(object):
14     """A Constraint consists of
15     * scope, a tuple of variables
16     * a condition, a function that can applied to a tuple of values
17     for the variables
18     """
19     def __init__(self, scope, condition):
20         self.scope = scope
21         self.condition = condition
22
23     def __repr__(self):
24         return self.condition.__name__ + str(self.scope)
```

An **assignment** is a variable:value dictionary. *con.holds(assignment)*, where *assignment* assigns a value to every variable in the scope of the constraint *con*,

returns True or False depending on whether the conditions is true or false in that assignment. This will give an error is not all variables in *con* are assigned.

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  $t$  have value  $(1,2,3)$ , then  $F(*t)$  is the same as  $F(1,2,3)$ .

```

_____cspProblem.py — (continued)_____
26     def holds(self,assignment):
27         """returns the value of Constraint con evaluated in assignment.
28
29         precondition: all variables are assigned in assignment
30         """
31         return self.condition(*tuple(assignment[v] for v in self.scope))

```

#### 4.1.2 CSPs

A constraint satisfaction problem consists of:

- a dictionary *domains* that maps variables to the set of possible values
- a set or list of constraints.

```

_____cspProblem.py — (continued)_____
33 class CSP(Displayable):
34     """A CSP consists of
35     * domains, a dictionary that maps each variable to its domain
36     * constraints, a list of constraints
37     """
38     def __init__(self,domains,constraints):
39         self.variables = set(domains)
40         self.domains = domains
41         self.constraints = constraints
42         self.var_to_const = {var:set() for var in self.variables}
43         for con in constraints:
44             for var in con.scope:
45                 self.var_to_const[var].add(con)
46
47     def __str__(self):
48         """string representation of CSP"""
49         return str(self.domains)
50
51     def __repr__(self):
52         """more detailed string representation of CSP"""
53         return "CSP("+str(self.domains)+", "+str([str(c) for c in self.constraints])+")"
54
55     def consistent(self,assignment):
56         """returns True if all of the constraints that can be evaluated
57         evaluate to True given the assignment.
58         assignment is a variable:value dictionary

```

```

59 |         """
60 |         return all(con.holds(assignment)
61 |                     for con in self.constraints
62 |                     if all(v in assignment for v in con.scope))

```

### 4.1.3 Examples

In the following code *ne\_*, when given a number, returns a function that is true when its argument is not that number. Thus if  $f = ne_ (3)$ , then  $f(2)$  is True and  $f(3)$  is False.

```

_____cspExamples.py — Example CSPs_____
11 | from cspProblem import CSP, Constraint
12 | from operator import lt,ne,eq,gt
13 |
14 | def ne_(val):
15 |     """not equal value"""
16 |     # return lambda x: x != val    # alternative definition
17 |     def nev(x):
18 |         return val != x
19 |     nev.__name__ = str(val)+"!="    # name of the function
20 |     return nev
21 |
22 | def is_(val):
23 |     """is a value"""
24 |     #return lambda x: x == val # alternative definition
25 |     # isv = partial(eq,val)    # another alternative definition
26 |     def isv(x):
27 |         return val == x
28 |     isv.__name__ = str(val)+"=="
29 |     return isv

```

The CSP, *csp0* has variables *A*, *B* and *C*, each with domain  $\{1,2,3\}$ . The constraints are  $A < B$  and  $B < C$ . This example is used in the unit tests, and so if it is changed, the unit tests need to be changed.

```

_____cspExamples.py — (continued)_____
31 | csp0 = CSP({'A':{1,2,3}, 'B':{1,2,3}, 'C':{1,2,3}},
32 |           [ Constraint(('A','B'),lt),
33 |             Constraint(('B','C'),lt)])

```

The CSP, *csp1* has variables *A*, *B* and *C*, each with domain  $\{1,2,3,4\}$ . The constraints are  $A < B$  and  $B < C$ . This is slightly more interesting than *csp0* as it has more solutions. Uncommenting the second constraint adds the constraint  $B \neq 2$ .

```

_____cspExamples.py — (continued)_____
35 | csp1 = CSP({'A':{1,2,3,4}, 'B':{1,2,3,4}, 'C':{1,2,3,4}},
36 |           [ Constraint(('A','B'),lt),
37 |             #Constraint(('B',),ne_(2)),    # alternatively: lambda b: b!=2

```

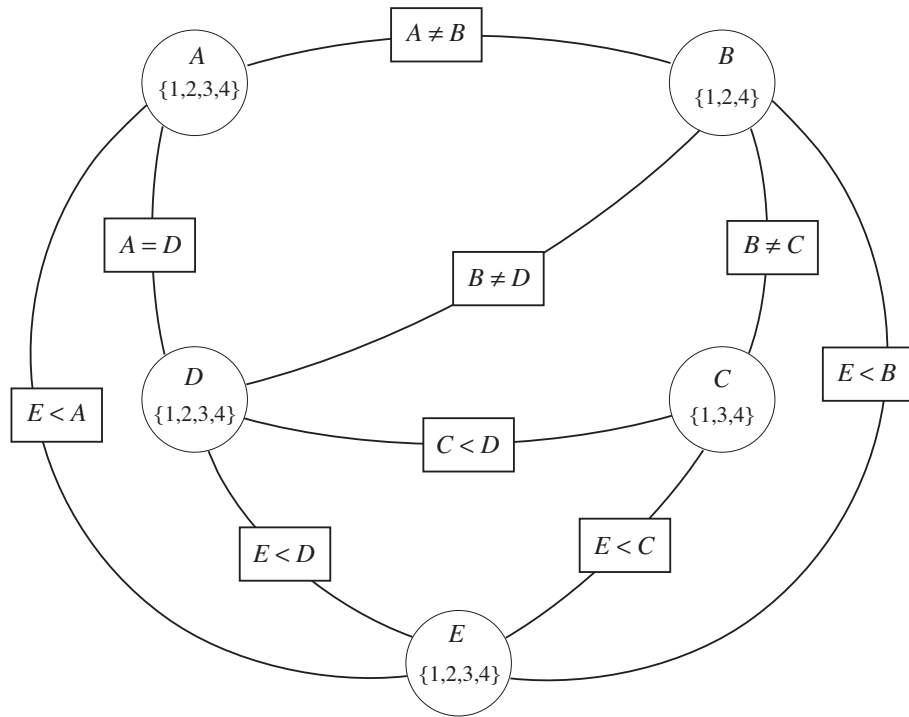


Figure 4.1: Domain-consistent constraint network.

```
38 | Constraint(('B', 'C'), lt))
```

The next CSP, *csp2* is Example 4.10 of the textbook; the domain consistent network is shown in Figure 4.1.

```

cspExamples.py — (continued)
40 | csp2 = CSP({'A':{1,2,3,4}, 'B':{1,2,3,4}, 'C':{1,2,3,4},
41 |          'D':{1,2,3,4}, 'E':{1,2,3,4}},
42 |          [ Constraint(('B',), ne_(3)),
43 |            Constraint(('C',), ne_(2)),
44 |            Constraint(('A', 'B'), ne),
45 |            Constraint(('B', 'C'), ne),
46 |            Constraint(('C', 'D'), lt),
47 |            Constraint(('A', 'D'), eq),
48 |            Constraint(('A', 'E'), gt),
49 |            Constraint(('B', 'E'), gt),
50 |            Constraint(('C', 'E'), gt),
51 |            Constraint(('D', 'E'), gt),
52 |            Constraint(('B', 'D'), ne)])
```

The following examples represent a crossword. The first is the crossword shown in Figure 4.2.

```
cspExamples.py — (continued)
```

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

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

```

54 def meet_at(p1,p2):
55     def meets(w1,w2): return w1[p1]==w2[p2]
56     meets.__name__ = "meet_at"+"'+str(p1)+'','"+str(p2)+''"
57     return meets
58
59 crossword1 = CSP({'one_across':{'ant', 'big', 'bus', 'car', 'has'},
60                  'one_down':{'book', 'buys', 'hold', 'lane', 'year'},
61                  'two_down':{'ginger', 'search', 'symbol', 'syntax'},
62                  'three_across':{'book', 'buys', 'hold', 'land', 'year'},
63                  'four_across':{'ant', 'big', 'bus', 'car', 'has'}},
64                  [Constraint(('one_across', 'one_down'),meet_at(0,0)),
65                    Constraint(('one_across', 'two_down'),meet_at(2,0)),
66                    Constraint(('three_across', 'two_down'),meet_at(2,2)),
67                    Constraint(('three_across', 'one_down'),meet_at(0,2)),
68                    Constraint(('four_across', 'two_down'),meet_at(0,4))])

```

The following examples represent a  $3 \times 3$  crossword, shown in Figure 4.3. Note that this is a superset of the words for Exercise 4.1 in the text. In the first representation, the variables represent words.

```

70 words1 = {"add", "ado", "age", "ago", "aid", "ail", "aim", "air",
71           "and", "any", "ape", "apt", "arc", "are", "ark", "arm", "art", "ash",
72           "ask", "auk", "awe", "awl", "aye", "bad", "bag", "ban", "bat", "bee",
73           "boa", "dim", "ear", "eel", "eft", "far", "fat", "fit", "lee", "oaf",
74           "rat", "tar", "tie"}
75 crossword2 = CSP({'1_down':words1, '2_down':words1, '3_down':words1,
76                  '1_across':words1, '4_across':words1, '5_across':words1},
77                  [Constraint(('1_down','1_across'),meet_at(0,0)), # 1_down[0]=1_across[0]
78                    Constraint(('1_down','4_across'),meet_at(1,0)), # 1_down[1]=4_across[0]
79                    Constraint(('1_down','5_across'),meet_at(2,0)),
80                    Constraint(('2_down','1_across'),meet_at(0,1)),
81                    Constraint(('2_down','4_across'),meet_at(1,1)),

```

1	2	3
4		
5		

**Words:**

add, age, aid, aim, air,  
are, arm, art, bad, bat,  
bee, boa, dim, ear, eel,  
eft, lee, oaf

Figure 4.3: A crossword puzzle to be solved with six words

```

82         Constraint(('2_down', '5_across'), meet_at(2,1)),
83         Constraint(('3_down', '1_across'), meet_at(0,2)),
84         Constraint(('3_down', '4_across'), meet_at(1,2)),
85         Constraint(('3_down', '5_across'), meet_at(2,2))
86     ])

```

The following representation of the crossword the variables represent letters.

```

cspExamples.py — (continued)
88 def is_word(*letters):
89     """is true if the letters concatenated forms a word in words1"""
90     return "".join(letters) in words1
91
92 letters = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
93           "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y",
94           "z"}
95 crossword2d = CSP({"p00":letters, "p01":letters, "p02":letters,
96                  "p10":letters, "p11":letters, "p12":letters,
97                  "p20":letters, "p21":letters, "p22":letters},
98                  [Constraint(("p00", "p01", "p02"), is_word),
99                    Constraint(("p10", "p11", "p12"), is_word),
100                    Constraint(("p20", "p21", "p22"), is_word),
101                    Constraint(("p00", "p10", "p20"), is_word),
102                    Constraint(("p01", "p11", "p21"), is_word),
103                    Constraint(("p02", "p12", "p22"), is_word)])

```

## Unit tests

The following defines a unit test for solvers on example csp0.

```

cspExamples.py — (continued)
105 def test(CSP_solver, csp=csp0, solutions=[{'A': 1, 'B': 2, 'C': 3}]):
106     """CSP_solver is a solver that finds a solution to a CSP.
107     CSP_solver takes a csp and returns a solution.
108     csp has to be a CSP, where solutions is the list of all solutions.
109     This tests whether the solution returned by CSP_solver is a solution.

```

```

110     """
111     print("Testing csp0 with",CSP_solver.__doc__)
112     sol0 = CSP_solver(csp)
113     print("Solution found:",sol0)
114     assert sol0 in solutions, "Solution not found for csp0"
115     print("Passed unit test")

```

## 4.2 Solving a CSP using Search

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

This searches through the space of partial assignments. This takes in a CSP problem and an optional variable ordering, which is a list of the variables in the CSP. A node in the search tree is a *variable : value* dictionary. This assumes a static ordering; the next variable chosen to split does not depend on the current context. If not 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 from utilities import dict_union
14
15 class Search_from_CSP(Search_problem):
16     """A search problem directly from the CSP.
17
18     A node is a variable:value dictionary"""
19     def __init__(self, csp, variable_order=None):
20         self.csp=csp
21         if variable_order:
22             assert set(variable_order) == set(csp.variables)
23             self.variables = variable_order
24         else:
25             self.variables = list(csp.variables)
26
27     def is_goal(self, node):
28         return len(node)==len(self.csp.variables)
29
30     def start_nodes(self):
31         return [{}]

```

Because we are not interested in the path, only the goal, and all paths to a goal has the same length, it makes little sense to use  $A^*$  or branch-and-bound. We can use the conceptually simpler depth-first search (Section 3.4), which requires an enumeration of the neighbor nodes.

The *neighbor\_nodes*(*node*) method enumerates the neighboring nodes of *node*. It uses that 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.

```

cspSearch.py — (continued)
33 def neighbor_nodes(self, node):
34     """iterator over the neighboring nodes of node"""
35     var = self.variables[len(node)] # the next variable
36     for val in self.csp.domains[var]:
37         new_env = dict_union(node,{var:val}) #dictionary union
38         if self.csp.consistent(new_env):
39             yield new_env

cspSearch.py — (continued)
41 from cspExamples import csp1,csp2,test
42 from searchDepthFirst import Depth_first_search
43
44 def dfs_solver(csp):
45     """depth-first search"""
46     return Depth_first_search(Search_from_CSP(csp)).search()
47
48 if __name__ == "__main__":
49     test(dfs_solver)
50
51 ## Test Solving CSPs with Search:
52 searcher1 = Depth_first_search(Search_from_CSP(csp1))
53 #print(searcher1.search()) # get next solution
54 searcher2 = Depth_first_search(Search_from_CSP(csp2))
55 #print(searcher2.search()) # get next solution

```

**Exercise 4.1** 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).

## 4.3 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. Because it changes the domains, it needs to make a copy of the domains to avoid side-effects.

```

cspConsistency.py — Arc Consistency and Domain splitting for solving a CSP
11 from utilities import Displayable

```



```

12
13 class Con_solver(Displayable):
14     def __init__(self, csp, domains=None):
15         self.csp=csp
16         if domains is not None:
17             self.domains = domains
18         else:
19             self.domains = csp.domains.copy()
20
21     def __repr__(self):
22         return str(self.domains)

```

This implementation of arc consistency maintains the set *to\_do* of (variable, constraint) pairs that are to be checked.

```

cspConsistency.py — (continued)
24 def make_arc_consistent(self, to_do=None):
25     """Makes this CSP arc-consistent using generalized arc consistency
26     to_do is a set of (variable,constraint) pairs
27     """
28     if to_do is None:
29         to_do = {(var,const) for const in self.csp.constraints
30                 for var in const.scope}
31     else:
32         to_do = to_do.copy() # use a copy of to_do
33     while to_do:
34         var,const = to_do.pop()
35         self.display(3,"Processing arc (",var,",",const,")")
36         other_vars = [ov for ov in const.scope if ov is not var]
37         new_domain = {val for val in self.domains[var]
38                       if self.any_holds(const,{var:val},other_vars,0)}
39         if new_domain != self.domains[var]:
40             self.display(2,"Domain pruned", "dom(",var,") =",new_domain," due to ",const)
41             self.domains[var] = new_domain
42             add_to_do = self.new_to_do(var,const)
43             to_do |= add_to_do # set union
44             self.display(3," adding",add_to_do if add_to_do else "nothing", "to to_do.")
45         self.display(2,"AC done. Reduced domains",self.domains)
46
47     def new_to_do(self, var, const):
48         """returns new elements to be added to to_do after assigning
49         variable var in constraint const.
50         """
51         return {(nvar,nconst) for nconst in self.csp.var_to_const[var]
52                 if nconst != const
53                 for nvar in nconst.scope
54                 if nvar != var}

```

*any\_holds* is a recursive function that tries to find an assignment of values to the other variables that satisfies constraint given the assignment in *env*. The variable *ind* specifies which index to *other\_vars* needs to be checked next. As

soon as one assignment returns *True*, the algorithm returns *True*. Note that it has side effects with respect to *env*; it should only be called when the side effects have no ill effects.

```

cspConsistency.py — (continued)
56 def any_holds(self, const, env, other_vars, ind):
57     """returns True if Constraint const holds for an assignment
58     that extends env with the variables in other_vars[ind:]
59     env is a dictionary
60     Warning: this has side effects and changes the elements of env
61     """
62     if ind==len(other_vars):
63         return const.holds(env)
64     else:
65         var = other_vars[ind]
66         for val in self.domains[var]:
67             #env = dict_union(env, {var:val}) # no side effects!
68             env[var] = val
69             if self.any_holds(const, env, other_vars, ind+1):
70                 return True
71     return False

```

Arc consistency has side effects of changing domains. The side effects are useful because they let us change the domains of each variable locally, without needing to copy all other variables. When carrying out arc consistency plus domain splitting, the side effects need to be local so they do not affect other branches of the search tree. Thus we need to make a copy of the domains. *copy\_with\_assign* takes a copy of the domains, perhaps allowing for a new domain for a variable.

This creates a copy of the CSP with an (optional) assignment of a new domain to a variable. Only the domains are copied.

```

cspConsistency.py — (continued)
73 def copy_with_assign(self, var=None, new_domain={True,False}):
74     """create a copy of the CSP with an assignment var=new_domain
75     if var==None then it is just a copy.
76     """
77     newdoms = self.domains.copy()
78     if var:
79         newdoms[var] = new_domain
80     return Con_solver(self.csp, newdoms)

```

### 4.3.1 Direct Implementation of Domain Splitting

The following is a direct implementation of domain splitting with arc consistency that uses recursion. It finds one solution if one exists or returns False if there are no solutions.

```

cspConsistency.py — (continued)

```

```

82
83 def solve_one(self,to_do=None):
84     """return a solution to the current CSP or False if there are no solutions
85     to_do is the list of arcs to check
86     """
87     self.make_arc_consistent(to_do)
88     if any(len(self.domains[var])==0 for var in self.domains):
89         return False
90     elif all(len(self.domains[var])==1 for var in self.domains):
91         self.display(2,"solution:", {var:select(self.domains[var]) for var in self.domains})
92         return {var:select(self.domains[var]) for var in self.domains}
93     else:
94         var = select(x for x in self.csp.variables if len(self.domains[x])>1)
95         if var:
96             split = len(self.domains[var])//2
97             dom1 = set(list(self.domains[var][:split])) #a nonempty proper subset
98             csp1 = self.copy_with_assign(var,dom1) # copy with dom(var)=dom1
99             dom2 = self.domains[var]-dom1
100             csp2 = self.copy_with_assign(var,dom2) # copy with dom(var)=dom2
101             self.display(3,"...splitting",var,"into",dom1,"and",dom2)
102             to_do = self.new_to_do(var,None)
103             return csp1.solve_one(to_do) or csp2.solve_one(to_do)

```

cspConsistency.py — (continued)

```

105 def select(iterable):
106     """select an element of iterable. Returns None if there is no such element.
107
108     This implementation just picks the first element.
109     For many of the uses, which element is selected does not affect correctness,
110     but may affect efficiency.
111     """
112     for e in iterable:
113         return e #returns first element found

```

**Exercise 4.2** Implement *solve\_all* that is like *solve\_one* but returns the set of all solution.

**Exercise 4.3** Implement *solve\_enum* that enumerates the solutions. It should use *yield* and *yield from*.

Unit test:

cspConsistency.py — (continued)

```

115 from cspExamples import test
116 def ac_solver(csp):
117     "arc consistency (solve_one)"
118     return Con_solver(csp).solve_one()
119 if __name__ == "__main__":
120     test(ac_solver)

```

### 4.3.2 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.

In the following we only implement *neighbor\_nodes* (used in depth-first search; Section 3.4) rather than *neighbors* (used in the other searchers). The following implements the standard protocol required for depth-first search.

A node is an instance of *Con\_solver*.

```

122 from searchProblem import Search_problem
123
124 class Search_with_AC_from_CSP(Search_problem, Displayable):
125     """A search problem with arc consistency and domain splitting
126
127     A node is a CSP """
128     def __init__(self, csp):
129         self.cons = Con_solver(csp) #copy of the CSP
130         self.cons.make_arc_consistent() # this has side effects
131
132     def is_goal(self, node):
133         """node is a goal if all domains have 1 element"""
134         return all(len(node.domains[var])==1 for var in node.domains)
135
136     def start_nodes(self):
137         return [self.cons]
138
139     def neighbor_nodes(self, node):
140         """an iterator over the neighboring nodes of node.
141         This is used for depth-first search"""
142         var = select(x for x in node.domains if len(node.domains[x])>1)
143         if var:
144             split = len(node.domains[var])//2
145             dom1 = set(list(node.domains[var][:split]) #a nonempty proper subset
146             dom2 = node.domains[var]-dom1
147             self.display(2, "Splitting", var, "into", dom1, "and", dom2)
148             to_do = node.new_to_do(var, None)
149             for dom in [dom1, dom2]:
150                 newcsp = node.copy_with_assign(var, dom)
151                 newcsp.make_arc_consistent(to_do)
152                 if all(len(newcsp.domains[v])>0 for v in newcsp.domains):
153                     # all domains are non-empty
154                     yield newcsp
155             else:
156                 self.display(2, "...", var, "in", dom, "has no solution")

```

**Exercise 4.4** 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?

**Exercise 4.5** Implement the protocol of  $A^*$  and the other searchers; this requires implementing *neighbors*. Does it make sense to use  $A^*$  search? What might be a good heuristic for  $A^*$ ?

Unit test:

```

cspConsistency.py — (continued)
158 from cspExamples import test
159 from searchDepthFirst import Depth_first_search
160
161 def ac_search_solver(csp):
162     """arc consistency (search interface)"""
163     sol = Depth_first_search(Search_with_AC_from_CSP(csp)).search().domains
164     return {v:select(d) for (v,d) in sol.items()}
165 if __name__ == "__main__":
166     test(ac_search_solver)

```

Testing:

```

cspConsistency.py — (continued)
168 from cspExamples import csp1, csp2, crossword1, crossword2, crossword2d
169
170 ## Test Solving CSPs with Arc consistency and domain splitting:
171 #Con_solver(csp1).solve_one()
172 #searcher1d = Depth_first_search(Search_with_AC_from_CSP(csp1))
173 #print(searcher1d.search().domains)
174 #Depth_first_search.max_display_level = 2 # display search trace (0 turns off)
175 #searcher2c = Depth_first_search(Search_with_AC_from_CSP(csp2))
176 #print(searcher2c.search().domains)
177 #searcher3c = Depth_first_search(Search_with_AC_from_CSP(crossword1))
178 #print(searcher3c.search().domains)
179 #searcher4c = Depth_first_search(Search_with_AC_from_CSP(crossword2))
180 #print(searcher4c.search().domains)
181 #searcher5c = Depth_first_search(Search_with_AC_from_CSP(crossword2d))
182 #print(searcher5c.search().domains)

```

## 4.4 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.

This implements both the two-stage choice, the any-conflict algorithm and a random choice of variable (and 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
11 from cspProblem import CSP, Constraint
12 from searchProblem import Arc, Search_problem
13 from utilities import Displayable
14 import random
15 import heapq
16
17 class SLSearcher(Displayable):
18     """A search problem directly from the CSP..
19
20     A node is a variable:value dictionary"""
21     def __init__(self, csp):
22         self.csp = csp
23         self.variables_to_select = {var for var in self.csp.variables
24                                     if len(self.csp.domains[var]) > 1}
25         # Create assignment and conflicts set
26         self.current_assignment = None # this will trigger a random restart
27         self.number_of_steps = 1 #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_sample(dom) for
33                               (var,dom) in self.csp.domains.items()}
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.

This method selects one of two implementations. When the probability of selecting a best variable (one involved in the most conflicts) 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*).

Note that for the probability parameters, any value less than 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 random variable is only chosen if *prob\_anycon* + *prob\_best* < 1.

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=1.0, prob_anycon=1.0):
43     """
44     returns the number of steps or None if there is no solution
45     if there is a solution, it can be found in self.current_assignment
46     """
47     if self.current_assignment is None:
48         self.restart()
49         self.number_of_steps += 1
50         if not self.conflicts:
51             return self.number_of_steps
52     if prob_best > 0: # we need to maintain a variable priority queue
53         return self.search_with_var_pq(max_steps, prob_best, prob_anycon)
54     else:
55         return self.search_with_any_conflict(max_steps, prob_anycon)

```

**Exercise 4.6** 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*).

#### 4.4.1 Any-conflict

If the probability of picking a best variable is zero, the implementation need to keep track of which variables are in conflicts.

```

cspSLS.py — (continued)
57 def search_with_any_conflict(self, max_steps, prob_anycon=1.0):
58     """Searches with the any_conflict heuristic.
59     This relies on just maintaining the set of conflicts;
60     it does not maintain a priority queue
61     """
62     self.variable_pq = None # we are not maintaining the priority queue.
63                             # This ensures it is regenerated if needed.
64     for i in range(max_steps):
65         self.number_of_steps += 1

```

```

66         if random.random() < probab_anycon:
67             con = random_sample(self.conflicts) # pick random conflict
68             var = random_sample(con.scope) # pick variable in conflict
69         else:
70             var = random_sample(self.variables_to_select)
71         if len(self.csp.domains[var]) > 1:
72             val = random_sample(self.csp.domains[var] -
73                               {self.current_assignment[var]})
74             self.display(2, "Assigning", var, "=", val)
75             self.current_assignment[var]=val
76             for varcon in self.csp.var_to_const[var]:
77                 if varcon.holds(self.current_assignment):
78                     if varcon in self.conflicts:
79                         self.conflicts.remove(varcon)
80                 else:
81                     if varcon not in self.conflicts:
82                         self.conflicts.add(varcon)
83             self.display(2, "Number of conflicts", len(self.conflicts))
84         if not self.conflicts:
85             self.display(1, "Solution found", self.current_assignment,
86                           "in", self.number_of_steps, "steps")
87         return self.number_of_steps
88     self.display(1, "No solution in", self.number_of_steps, "steps",
89                 len(self.conflicts), "conflicts remain")
90     return None

```

**Exercise 4.7** This makes no attempt to find the best alternative value for a variable. Modify the code so that after selecting a variable it selects a value that reduces the number of conflicts by the most. Have a parameter that specifies the probability that the best value is chosen.

#### 4.4.2 Two-Stage Choice

This is the top-level searching algorithm that maintains a priority queue of variables ordered by (the negative of) 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. This uses the dictionary *var\_differential* which specifies how much the values of variables should change. This is used with the updatable queue (page 66) to find a variable with the most conflicts.

```

cspSLS.py — (continued)
92     def search_with_var_pq(self, max_steps, probab_best=1.0, probab_anycon=1.0):
93         """search with a priority queue of variables.
94         This is used to select a variable with the most conflicts.
95         """
96         if not self.variable_pq:
97             self.create_pq()

```



```

98     pick_best_or_con = prob_best + prob_anycon
99     for i in range(max_steps):
100         self.number_of_steps += 1
101         randnum = random.random()
102         ## Pick a variable
103         if randnum < prob_best: # pick best variable
104             var,oldval = self.variable_pq.top()
105         elif randnum < pick_best_or_con: # pick a variable in a conflict
106             con = random_sample(self.conflicts)
107             var = random_sample(con.scope)
108         else: #pick any variable that can be selected
109             var = random_sample(self.variables_to_select)
110         if len(self.csp.domains[var]) > 1: # var has other values
111             ## Pick a value
112             val = random_sample(self.csp.domains[var] -
113                                {self.current_assignment[var]})
114             self.display(2,"Assigning",var,val)
115             ## Update the priority queue
116             var_differential = {}
117             self.current_assignment[var]=val
118             for varcon in self.csp.var_to_const[var]:
119                 self.display(3,"Checking",varcon)
120                 if varcon.holds(self.current_assignment):
121                     if varcon in self.conflicts: #was incons, now consis
122                         self.display(3,"Became consistent",varcon)
123                         self.conflicts.remove(varcon)
124                         for v in varcon.scope: # v is in one fewer conflicts
125                             var_differential[v] = var_differential.get(v,0)-1
126                 else:
127                     if varcon not in self.conflicts: # was consis, not now
128                         self.display(3,"Became inconsistent",varcon)
129                         self.conflicts.add(varcon)
130                         for v in varcon.scope: # v is in one more conflicts
131                             var_differential[v] = var_differential.get(v,0)+1
132             self.variable_pq.update_each_priority(var_differential)
133             self.display(2,"Number of conflicts",len(self.conflicts))
134             if not self.conflicts: #no conflicts, so solution found
135                 self.display(1,"Solution found",self.current_assignment,"in",
136                             self.number_of_steps,"steps")
137                 return self.number_of_steps
138             self.display(1,"No solution in",self.number_of_steps,"steps",
139                         len(self.conflicts),"conflicts remain")
140         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.

```

142     def create_pq(self):
143         """Create the variable to number-of-conflicts priority queue.
144         This is needed to select the variable in the most conflicts.
145
146         The value of a variable in the priority queue is the negative of the
147         number of conflicts the variable appears in.
148         """
149         self.variable_pq = Updatable_priority_queue()
150         var_to_number_conflicts = {}
151         for con in self.conflicts:
152             for var in con.scope:
153                 var_to_number_conflicts[var] = var_to_number_conflicts.get(var,0)+1
154         for var,num in var_to_number_conflicts.items():
155             if num>0:
156                 self.variable_pq.add(var,-num)

```

cspSLS.py — (continued)

```

158     def random_sample(st):
159         """selects a random element from set st"""
160         return random.sample(st,1)[0]

```

**Exercise 4.8** This makes no attempt to find the best alternative value for a variable. Modify the code so that after selecting a variable it selects a value that reduces the number of conflicts by the most. Have a parameter that specifies the probability that the best value is chosen.

**Exercise 4.9** 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)?

### 4.4.3 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/3.5/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)

```

162 class Updatable_priority_queue(object):

```

```

163 """A priority queue where the values can be updated.
164 Elements with the same value are ordered randomly.
165
166 This code is based on the ideas described in
167 http://docs.python.org/3.3/library/heapq.html
168 It could probably be done more efficiently by
169 shuffling the modified element in the heap.
170 """
171 def __init__(self):
172     self.pq = [] # priority queue of [val,rand,elt] triples
173     self.elt_map = {} # map from elt to [val,rand,elt] triple in pq
174     self.REMOVED = "*removed*" # a string that won't be a legal element
175     self.max_size=0
176
177 def add(self,elt,val):
178     """adds elt to the priority queue with priority=val.
179     """
180     assert val <= 0, val
181     assert elt not in self.elt_map, elt
182     new_triple = [val, random.random(), elt]
183     heapq.heappush(self.pq, new_triple)
184     self.elt_map[elt] = new_triple
185
186 def remove(self,elt):
187     """remove the element from the priority queue"""
188     if elt in self.elt_map:
189         self.elt_map[elt][2] = self.REMOVED
190         del self.elt_map[elt]
191
192 def update_each_priority(self,update_dict):
193     """update values in the priority queue by subtracting the values in
194     update_dict from the priority of those elements in priority queue.
195     """
196     for elt,incr in update_dict.items():
197         if incr != 0:
198             newval = self.elt_map.get(elt,[0])[0] - incr
199             assert newval <= 0, str(elt)+": "+str(newval+incr)+"-"+str(incr)
200             self.remove(elt)
201             if newval != 0:
202                 self.add(elt,newval)
203
204 def pop(self):
205     """Removes and returns the (elt,value) pair with minimal value.
206     If the priority queue is empty, IndexError is raised.
207     """
208     self.max_size = max(self.max_size, len(self.pq)) # keep statistics
209     triple = heapq.heappop(self.pq)
210     while triple[2] == self.REMOVED:
211         triple = heapq.heappop(self.pq)
212     del self.elt_map[triple[2]]

```

```

213         return triple[2], triple[0] # elt, value
214
215     def top(self):
216         """Returns the (elt,value) pair with minimal value, without removing it.
217         If the priority queue is empty, IndexError is raised.
218         """
219         self.max_size = max(self.max_size, len(self.pq)) # keep statistics
220         triple = self.pq[0]
221         while triple[2] == self.REMOVED:
222             heapq.heappop(self.pq)
223             triple = self.pq[0]
224         return triple[2], triple[0] # elt, value
225
226     def empty(self):
227         """returns True iff the priority queue is empty"""
228         return all(triple[2] == self.REMOVED for triple in self.pq)

```

#### 4.4.4 Plotting Runtime Distributions

*Runtime\_distribution* uses matplotlib to plot runtime distributions. Here the runtime is a misnomer as we are only plotting the number of steps, not the time. Computing the runtime is non-trivial as many of the runs have a very short runtime. 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 runtime. This is left as an exercise.

```

_____cspSLS.py — (continued)_____
230 import matplotlib.pyplot as plt
231
232 class Runtime_distribution(object):
233     def __init__(self, csp, xscale='log'):
234         """Sets up plotting for csp
235         xscale is either 'linear' or 'log'
236         """
237         self.csp = csp
238         plt.ion()
239         plt.xlabel("Number of Steps")
240         plt.ylabel("Cumulative Number of Runs")
241         plt.xscale(xscale) # Makes a 'log' or 'linear' scale
242
243     def plot_run(self, num_runs=100, max_steps=1000, prob_best=1.0, prob_anycon=1.0):
244         stats = []
245         SLSearcher.max_display_level, temp_mdl = 0, SLSearcher.max_display_level # no display
246         for i in range(num_runs):
247             searcher = SLSearcher(self.csp)
248             num_steps = searcher.search(max_steps, prob_best, prob_anycon)
249             if num_steps:
250                 stats.append(num_steps)
251         stats.sort()

```

```

252         if prob_best >= 1.0:
253             label = "P(best)=1.0"
254         else:
255             p_ac = min(prob_anycon, 1-prob_best)
256             label = "P(best)=%.2f, P(ac)=%.2f" % (prob_best, p_ac)
257         plt.plot(stats, range(len(stats)), label=label)
258         plt.legend(loc="upper left")
259         #plt.draw()
260         SLSearcher.max_display_level= temp_mdl #restore display

```

### 4.4.5 Testing

```

cspSLS.py — (continued)
262 from cspExamples import test
263 def sls_solver(csp, prob_best=0.7):
264     """stochastic local searcher"""
265     se0 = SLSearcher(csp)
266     se0.search(1000, prob_best)
267     return se0.current_assignment
268 def any_conflict_solver(csp):
269     """stochastic local searcher (any-conflict)"""
270     return sls_solver(csp, 0)
271
272 if __name__ == "__main__":
273     test(sls_solver)
274     test(any_conflict_solver)
275
276 from cspExamples import csp1, csp2, crossword1
277
278 ## Test Solving CSPs with Search:
279 #se1 = SLSearcher(csp1); print(se1.search(100))
280 #se2 = SLSearcher(csp2); print(se2.search(1000, 1.0)) # greedy
281 #se2 = SLSearcher(csp2); print(se2.search(1000, 0)) # any_conflict
282 #se2 = SLSearcher(csp2); print(se2.search(1000, 0.7)) # 70% greedy; 30% any_conflict
283 #SLSearcher.max_display_level=2 #more detailed display
284 #se3 = SLSearcher(crossword1); print(se3.search(100), 0.7)
285 #p = Runtime_distribution(csp2)
286 #p.plot_run(100, 1000, 0)
287 #p.plot_run(100, 1000, 1.0)
288 #p.plot_run(100, 1000, 0.7)

```

**Exercise 4.10** Modify this to plot the runtime, instead of the number of steps. To measure runtime use *timeit* (<https://docs.python.org/3.5/library/timeit.html>). Small runtimes are inaccurate, so *timeit* can run the same code multiple times. Stochastic local algorithms give different runtimes 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/3.5/library/random.html>). Because the runtime 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.

## 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.

```
_____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 __str__(self):
20         """returns the string representation of a clause.
21         """
22         if self.body:
23             return self.head + " <- " + " & ".join(self.body) + "."
24         else:
25             return 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 "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', 'yes.', 'oui', 'oui.', 'y', 'y.'] # bilingual

```

A knowledge base is a list of clauses and askables. In order to make top-down inference faster, this creates a dictionary that maps each atoms into the set of clauses with that atom in the head.

---

```

42 from utilities 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             if c.head in self.atom_to_clauses:
55                 self.atom_to_clauses[c.head].add(c)
56             else:
57                 self.atom_to_clauses[c.head] = {c}
58
59     def clauses_for_atom(self,a):
60         """returns set of clauses with atom a as the head"""
61         if a in self.atom_to_clauses:
62             return self.atom_to_clauses[a]
63         else:
64             return set()
65
66     def __str__(self):
67         """returns a string representation of this knowledge base.
68         """
69         return '\n'.join([str(c) for c in self.statements])

```

---

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

---

```

71 triv_KB = KB([
72     Clause('i_am', ['i_think']),
73     Clause('i_think'),
74     Clause('i_smell', ['i_exist'])
75 ])

```

---



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

```

logicProblem.py — (continued)
77 elect = KB([
78     Clause('light_l1'),
79     Clause('light_l2'),
80     Clause('ok_l1'),
81     Clause('ok_l2'),
82     Clause('ok_cb1'),
83     Clause('ok_cb2'),
84     Clause('live_outside'),
85     Clause('live_l1', ['live_w0']),
86     Clause('live_w0', ['up_s2', 'live_w1']),
87     Clause('live_w0', ['down_s2', 'live_w2']),
88     Clause('live_w1', ['up_s1', 'live_w3']),
89     Clause('live_w2', ['down_s1', 'live_w3']),
90     Clause('live_l2', ['live_w4']),
91     Clause('live_w4', ['up_s3', 'live_w3']),
92     Clause('live_p_1', ['live_w3']),
93     Clause('live_w3', ['live_w5', 'ok_cb1']),
94     Clause('live_p_2', ['live_w6']),
95     Clause('live_w6', ['live_w5', 'ok_cb2']),
96     Clause('live_w5', ['live_outside']),
97     Clause('lit_l1', ['light_l1', 'live_l1', 'ok_l1']),
98     Clause('lit_l2', ['light_l2', 'live_l2', 'ok_l2']),
99     Askable('up_s1'),
100    Askable('down_s1'),
101    Askable('up_s2'),
102    Askable('down_s2'),
103    Askable('up_s3'),
104    Askable('down_s2')
105 ])
106
107 # print(kb)

```

## 5.2 Bottom-up Proofs

*fixed\_point* computes the fixed point of the knowledge base *kb*.

```

logicBottomUp.py — Bottom-up Proof Procedure for Definite Clauses
11 from logicProblem import yes
12
13 def fixed_point(kb):
14     """Returns the fixed point of knowledge base kb.
15     """
16     fp = ask_askables(kb)
17     added = True
18     while added:
19         added = False # added is true when an atom was added to fp this iteration

```

```

20     for c in kb.clauses:
21         if c.head not in fp and all(b in fp for b in c.body):
22             fp.add(c.head)
23             added = True
24             kb.display(2,c.head,"added to fp due to clause",c)
25     return fp
26
27 def ask_askables(kb):
28     return {at for at in kb.askables if yes(input("Is "+at+" true? "))}

```

Testing:

```

_____logicBottomUp.py — (continued)_____
30 from logicProblem import triv_KB
31 def test():
32     fp = fixed_point(triv_KB)
33     assert fp == {'i_am','i_think'}, "triv_KB gave result "+str(fp)
34     print("Passed unit test")
35 if __name__ == "__main__":
36     test()
37
38 from logicProblem import elect
39 # elect.max_display_level=3 # give detailed trace
40 # 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  $e$ .

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 asymptotically more efficient (in terms of  $b$ ) 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  $pf$ , 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

*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 tracing the code (and doesn't need to have a non-default value).

```

_____logicTopDown.py — Top-down Proof Procedure for Definite Clauses_____
11 from logicProblem import yes
12
13 def prove(kb, ans_body, indent=""):
14     """returns True if kb |- ans_body
15     """
16     kb.display(2, indent, 'yes <-', ' & '.join(ans_body))
17     if ans_body:
18         selected = ans_body[0] # select first atom from ans_body
19         if selected in kb.askables:
20             return (yes(input("Is "+selected+" true? "))
21                     and prove(kb, ans_body[1:], indent+" "))
22         else:
23             return any(prove(kb, cl.body+ans_body[1:], indent+" ")
24                         for cl in kb.clauses_for_atom(selected))
25     else:
26         return True # empty body is true

```

Testing:

```

_____logicTopDown.py — (continued)_____
28 from logicProblem import triv_KB
29 def test():
30     a1 = prove(triv_KB, ['i_am'])
31     assert a1, "triv_KB proving i_am gave "+str(a1)
32     a2 = prove(triv_KB, ['i_smell'])
33     assert not a2, "triv_KB proving i_smell gave "+str(a2)
34     print("Passed unit tests")
35 if __name__ == "__main__":
36     test()
37 # try
38 from logicProblem import elect
39 # elect.max_display_level=3 # give detailed trace
40 # prove(elect, ['live_w6'])
41 # 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.

## 5.4 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*.

```

11 from logicProblem import Clause, Askable, KB, yes
12
13 class Assumable(object):
14     """An askable atom"""
15
16     def __init__(self, atom):
17         """clause with atom head and lost of atoms body"""
18         self.atom = atom
19
20     def __str__(self):
21         """returns the string representation of a clause.
22         """
23         return "assumable " + self.atom + "."
24
25 class KBA(KB):
26     """A knowledge base that can include assumables"""
27     def __init__(self, statements):
28         self.assumables = [c.atom for c in statements if isinstance(c, Assumable)]
29         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.

```

31 def prove_all_ass(self, ans_body, assumed=set()):
32     """returns a list of sets of assumables that extends assumed
33     to imply ans_body from self.
34     ans_body is a list of atoms (it is the body of the answer clause).
35     assumed is a set of assumables already assumed
36     """
37     if ans_body:
38         selected = ans_body[0] # select first atom from ans_body
39         if selected in self.askables:
40             if yes(input("Is "+selected+" true? ")):
41                 return self.prove_all_ass(ans_body[1:], assumed)
42             else:
43                 return [] # no answers
44         elif selected in self.assumables:
45             return self.prove_all_ass(ans_body[1:], assumed|{selected})
46         else:
47             return [ass
48                     for cl in self.clauses_for_atom(selected)
49                     for ass in self.prove_all_ass(cl.body+ans_body[1:], assumed)
50                     ] # union of answers for each clause with head=selected
51     else:
52         return [assumed] # empty body
53

```

```

54 | def conflicts(self):
55 |     """returns a list of minimal conflicts"""
56 |     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)_____
58 | def minsets(ls):
59 |     """ls is a list of sets
60 |     returns a list of minimal sets in ls
61 |     """
62 |     ans = [] # elements known to be minimal
63 |     for c in ls:
64 |         if not any(c1 < c for c1 in ls) and not any(c1 <= c for c1 in ans):
65 |             ans.append(c)
66 |     return ans
67 |
68 | # 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. 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)_____
69 | def diagnoses(cons):
70 |     """cons is a list of (minimal) conflicts.
71 |     returns a list of diagnoses."""
72 |     if cons == []:
73 |         return [set()]
74 |     else:
75 |         return minsets([(e|d) # | is set union
76 |                         for e in cons[0]
77 |                         for d in diagnoses(cons[1:])])

```

Test cases:

```

_____logicAssumables.py — (continued)_____
80 | electa = KBA([
81 |     Clause('light_l1'),
82 |     Clause('light_l2'),
83 |     Assumable('ok_l1'),
84 |     Assumable('ok_l2'),
85 |     Assumable('ok_s1'),
86 |     Assumable('ok_s2'),
87 |     Assumable('ok_s3'),
88 |     Assumable('ok_cb1'),
89 |     Assumable('ok_cb2'),

```

```

90     Assumable('live_outside'),
91     Clause('live_l1', ['live_w0']),
92     Clause('live_w0', ['up_s2', 'ok_s2', 'live_w1']),
93     Clause('live_w0', ['down_s2', 'ok_s2', 'live_w2']),
94     Clause('live_w1', ['up_s1', 'ok_s1', 'live_w3']),
95     Clause('live_w2', ['down_s1', 'ok_s1', 'live_w3' ]),
96     Clause('live_l2', ['live_w4']),
97     Clause('live_w4', ['up_s3', 'ok_s3', 'live_w3' ]),
98     Clause('live_p1', ['live_w3']),
99     Clause('live_w3', ['live_w5', 'ok_cb1']),
100    Clause('live_p2', ['live_w6']),
101    Clause('live_w6', ['live_w5', 'ok_cb2']),
102    Clause('live_w5', ['live_outside']),
103    Clause('lit_l1', ['light_l1', 'live_l1', 'ok_l1']),
104    Clause('lit_l2', ['light_l2', 'live_l2', 'ok_l2']),
105    Askable('up_s1'),
106    Askable('down_s1'),
107    Askable('up_s2'),
108    Askable('down_s2'),
109    Askable('up_s3'),
110    Askable('down_s2'),
111    Askable('dark_l1'),
112    Askable('dark_l2'),
113    Clause('false', ['dark_l1', 'lit_l1']),
114    Clause('false', ['dark_l2', 'lit_l2'])
115    ])
116 # electa.prove_all_ass(['false'])
117 # cs=electa.conflicts()
118 # print(cs)
119 # diagnoses(cs)          # diagnoses from conflicts

```

**Exercise 5.5** 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 5.6** Implement *explanations(self, body)*, where *body* is a list of atoms, that returns the a list of the minimal explanations of the body. This does not require modification of *prove\_all\_ass*.

**Exercise 5.7** 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.

## Planning with Certainty

### 6.1 Representing Actions and Planning Problems

The STRIPS representation of an action consists of:

- 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.

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

A STRIPS domain consists of:

- A set of actions.

- A dictionary that maps each feature into a set of possible values for the feature.
- A dictionary that maps each action into a STRIPS representation of the action.

```

stripsProblem.py — (continued)
24 class STRIPS_domain(object):
25     def __init__(self, feats_vals, strips_map):
26         """Problem domain
27         feats_vals is a feature:domain dictionary,
28         mapping each feature to its domain
29         strips_map is an action:strips dictionary,
30         mapping each action to its Strips representation
31         """
32         self.actions = set(strips_map) # set of all actions
33         self.feats_vals = feats_vals
34         self.strips_map = strips_map

```

### 6.1.1 Robot Delivery Domain

The following specifies the robot delivery domain of Chapter 8.

```

stripsProblem.py — (continued)
36 boolean = {True, False}
37 delivery_domain = STRIPS_domain(
38     {'RLoc':{'cs', 'off', 'lab', 'mr'}, 'RHC':boolean, 'SWC':boolean,
39     'MW':boolean, 'RHM':boolean}, #features:values dictionary
40     {'mc_cs': Strips({'RLoc':'cs'}, {'RLoc':'off'}),
41     'mc_off': Strips({'RLoc':'off'}, {'RLoc':'lab'}),
42     'mc_lab': Strips({'RLoc':'lab'}, {'RLoc':'mr'}),
43     'mc_mr': Strips({'RLoc':'mr'}, {'RLoc':'cs'}),
44     'mcc_cs': Strips({'RLoc':'cs'}, {'RLoc':'mr'}),
45     'mcc_off': Strips({'RLoc':'off'}, {'RLoc':'cs'}),
46     'mcc_lab': Strips({'RLoc':'lab'}, {'RLoc':'off'}),
47     'mcc_mr': Strips({'RLoc':'mr'}, {'RLoc':'lab'}),
48     'puc': Strips({'RLoc':'cs', 'RHC':False}, {'RHC':True}),
49     'dc': Strips({'RLoc':'off', 'RHC':True}, {'RHC':False, 'SWC':False}),
50     'pum': Strips({'RLoc':'mr', 'MW':True}, {'RHM':True, 'MW':False}),
51     'dm': Strips({'RLoc':'off', 'RHM':True}, {'RHM':False})
52     } )

```

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)
54 class Planning_problem(object):
55     def __init__(self, prob_domain, initial_state, goal):
56         """

```



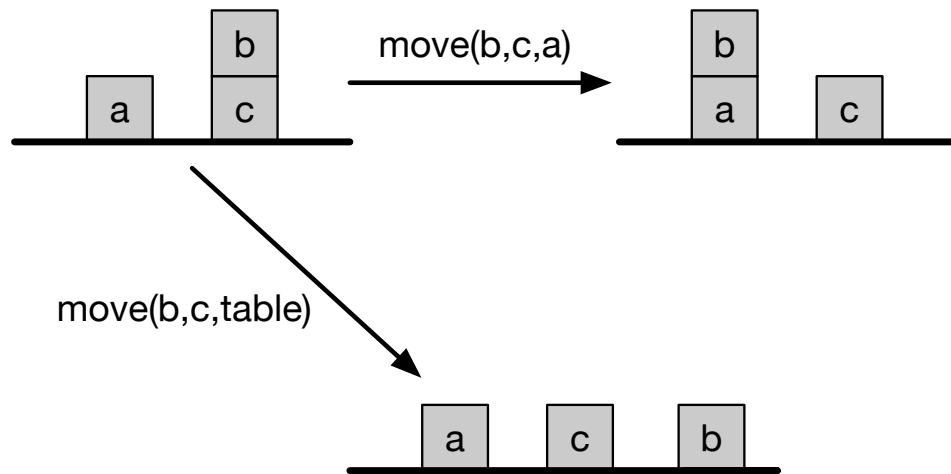


Figure 6.1: Blocks world with two actions

```

57     a planning problem consists of
58     * a planning domain
59     * the initial state
60     * a goal
61     """
62     self.prob_domain = prob_domain
63     self.initial_state = initial_state
64     self.goal = goal
65
66 problem0 = Planning_problem(delivery_domain,
67                             {'RLoc':'lab', 'MW':True, 'SWC':True, 'RHC':False,
68                              'RHM':False},
69                             {'RLoc':'off'})
70 problem1 = Planning_problem(delivery_domain,
71                             {'RLoc':'lab', 'MW':True, 'SWC':True, 'RHC':False,
72                              'RHM':False},
73                             {'SWC':False})
74 problem2 = Planning_problem(delivery_domain,
75                             {'RLoc':'lab', 'MW':True, 'SWC':True, 'RHC':False,
76                              'RHM':False},
77                             {'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 6.1 shows 3 states with some of the actions between them. The following represents the blocks world. Note that the actions and the conditions are all strings.

```

stripsProblem.py — (continued)
79  """ blocks world
80  def move(x,y,z):
81      """string for the move action"""
82      return 'move_'+x+'_from_'+y+'_to_'+z
83  def on(x,y):
84      """string for the 'on' feature"""
85      return x+'_on_'+y
86  def clear(x):
87      """string for the 'clear' feature"""
88      return 'clear_'+x
89  def create_blocks_world(blocks = ['a','b','c','d']):
90      blocks_and_table = blocks+['table']
91      stmap = {move(x,y,z):Strips({on(x,y):True, clear(x):True, clear(z):True},
92                                {on(x,z):True, on(x,y):False, clear(y):True, clear(z):False})
93              for x in blocks
94              for y in blocks_and_table
95              for z in blocks
96              if x!=y and y!=z and z!=x}
97      stmap.update({move(x,y,'table'):Strips({on(x,y):True, clear(x):True},
98                                             {on(x,'table'):True, on(x,y):False, clear(y):True})
99                  for x in blocks
100                 for y in blocks
101                 if x!=y})
102      feats_vals = {on(x,y):boolean for x in blocks for y in blocks_and_table}
103      feats_vals.update({clear(x):boolean for x in blocks_and_table})
104      return STRIPS_domain(feats_vals, stmap)

```

This is a classic example, with 3 blocks, and the goal consists of two conditions.

```

stripsProblem.py — (continued)
106 blocks1dom = create_blocks_world(['a','b','c'])
107 blocks1 = Planning_problem(blocks1dom,
108     {on('a','table'):True,clear('a'):True, clear('b'):True,on('b','c'):True,
109     on('c','table'):True,clear('c'):False}, # initial state
110     {on('a','b'):True, on('c','a'):True}) #goal

```

This is a problem of inverting a tower of size 4.

```

stripsProblem.py — (continued)
112 blocks2dom = create_blocks_world(['a','b','c','d'])
113 tower4 = {clear('a'):True, on('a','b'):True, clear('b'):False,
114           on('b','c'):True, clear('c'):False, on('c','d'):True,
115           clear('b'):False, on('d','table'):True}
116 blocks2 = Planning_problem(blocks2dom,
117     tower4, # initial state
118     {on('d','c'):True,on('c','b'):True,on('b','a'):True}) #goal

```

Moving bottom block to top of a tower of size 4.

```

stripsProblem.py — (continued)
120 blocks3 = Planning_problem(blocks2dom,

```

```

121 |     tower4, # initial state
122 |     {on('d','a'):True, on('a','b'):True, on('b','c'):True}) #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** The representation of the state does not include negative *on* facts. Does it need to? 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? (Does this change an answer to the previous question?)

## 6.2 Forward Planner

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, which is a variable:value dictionary. In order to be able to do multiple-path pruning, we need to define a hash function, and equality between states.

```

stripsForwardPlanner.py — Forward Planner with STRIPS actions
11 | from searchProblem import Arc, Search_problem
12 | from stripsProblem import Strips, STRIPS_domain
13 |
14 | class State(object):
15 |     def __init__(self, assignment):
16 |         self.assignment = assignment
17 |         self.hash_value = None
18 |     def __hash__(self):
19 |         if self.hash_value is None:
20 |             self.hash_value = hash(frozenset(self.assignment.items()))
21 |         return self.hash_value
22 |     def __eq__(self, st):
23 |         return self.assignment == st.assignment
24 |     def __str__(self):
25 |         return str(self.assignment)

```

In order to define a search problem (page 31), we need to define the goal condition, the start nodes, the neighbours, and (optionally) a heuristic function. Here *zero* is the default heuristic function.

stripsForwardPlanner.py — (continued)

```

27 def zero(*args,**nargs):
28     """always returns 0"""
29     return 0
30
31 class Forward_STRIPs(Search_problem):
32     """A search problem from a planning problem where:
33     * a node is a state object.
34     * the dynamics are specified by the STRIPS representation of actions
35     """
36     def __init__(self, planning_problem, heur=zero):
37         """creates a forward search space from a planning problem.
38         heur(state,goal) is a heuristic function,
39         an underestimate of the cost from state to goal, where
40         both state and goals are feature:value dictionaries.
41         """
42         self.prob_domain = planning_problem.prob_domain
43         self.initial_state = State(planning_problem.initial_state)
44         self.goal = planning_problem.goal
45         self.heur = heur
46
47     def is_goal(self, state):
48         """is True if node is a goal.
49
50         Every goal feature has the same value in the state and the goal."""
51         state_asst = state.assignment
52         return all(prop in state_asst and state_asst[prop]==self.goal[prop]
53                   for prop in self.goal)
54
55     def start_nodes(self):
56         """returns list of start nodes"""
57         return [self.initial_state]
58
59     def neighbors(self,state):
60         """returns neighbors of state in this problem"""
61         cost=1
62         state_asst = state.assignment
63         return [ Arc(state,self.effect(act,state_asst),cost,act)
64                 for act in self.prob_domain.actions
65                 if self.possible(act,state_asst)]
66
67     def possible(self,act,state_asst):
68         """True if act is possible in state.
69         act is possible if all of its preconditions have the same value in the state"""
70         preconds = self.prob_domain.strips_map[act].preconditions
71         return all(pre in state_asst and state_asst[pre]==preconds[pre]
72                   for pre in preconds)
73
74     def effect(self,act,state_asst):
75         """returns the state that is the effect of doing act given state_asst"""

```

```

76         new_state_asst = self.prob_domain.strips_map[act].effects.copy()
77         for prop in state_asst:
78             if prop not in new_state_asst:
79                 new_state_asst[prop]=state_asst[prop]
80         return State(new_state_asst)
81
82     def heuristic(self,state):
83         """in the forward planner a node is a state.
84         the heuristic is an (under)estimate of the cost
85         of going from the state to the top-level goal.
86         """
87         return self.heur(state.assignment, self.goal)

```

Here are some test cases to try.

```

_____stripsForwardPlanner.py — (continued) _____
89 from searchBranchAndBound import DF_branch_and_bound
90 from searchAStar import Searcher
91 from searchMPP import SearcherMPP
92 from stripsProblem import problem0, problem1, problem2, blocks1, blocks2, blocks3
93
94 # Searcher(Forward_STRIPS(problem1)).search() #A*
95 # SearcherMPP(Forward_STRIPS(problem1)).search() #A* with MPP
96 # DF_branch_and_bound(Forward_STRIPS(problem1),10).search() #B&B
97 # To find more than one plan:
98 # s1 =Searcher(Forward_STRIPS(problem1)) #A*
99 # 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 a (not very good) heuristic for the coffee delivery planning domain.

Note that this assumes 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, it needs to define whether the feature is defined in the goal.

```

_____stripsHeuristic.py — Planner with Heuristic Function _____
11 def heuristic_fun(state,goal):
12     """An (under)estimate of the cost of solving goal from state.
13     Both state and goal are variable:value dictionaries.
14     This heuristic is the maximum of
15     - the distance to the goal location, if there is one
16     - the distance to the coffee shop if the goal has SWC=False, node has SWC=True and RHC=False
17     """

```

```

18     return max(h1(state,goal),
19                h2(state,goal))
20
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 goal has SWC=False, node has SWC=True and RHC=False
31     """
32     if 'SWC' in goal and goal['SWC']==False and state['SWC']==True and state['RHC']==False:
33         return dist(state['RLoc'],'cs')+3
34     else:
35         return 0
36
37 def dist(loc1, loc2):
38     """returns the distance from location loc1 to loc2
39     """
40     if loc1==loc2:
41         return 0
42     if {loc1,loc2} in [{'cs','lab'},{'mr','off'}]:
43         return 2
44     else:
45         return 1

```

The following runs the example with and without the heuristic. (Also try using *Searcher* instead if *SearcherMPP*.)

```

stripsHeuristic.py — (continued)
47 ##### Forward Planner #####
48 from searchAStar import Searcher
49 from searchMPP import SearcherMPP
50 from stripsForwardPlanner import Forward_STRIPS
51 from stripsProblem import problem0, problem1, problem2
52 thisproblem = problem1
53
54 print("\n***** FORWARD NO HEURISTIC")
55 print(SearcherMPP(Forward_STRIPS(thisproblem)).search())
56
57 print("\n***** FORWARD WITH HEURISTIC")
58 print(SearcherMPP(Forward_STRIPS(thisproblem,heuristic_fun)).search())

```

**Exercise 6.4** Try the regression planner with a heuristic function of just *h1* and with just *h2*. Explain how each one prunes or doesn't prune the search space.

**Exercise 6.5** Create a better heuristic than *heuristic\_fun*. Try it for a number of different problems.

**Exercise 6.6** Create an admissible heuristic for the blocks world.

## 6.3 Regression Planner

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* object consists of an assignment, which is *variable:value* dictionary. We make it 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_____
11 from searchProblem import Arc, Search_problem
12
13 class Subgoal(object):
14     def __init__(self, assignment):
15         self.assignment = assignment
16         self.hash_value = None
17     def __hash__(self):
18         if self.hash_value is None:
19             self.hash_value = hash(frozenset(self.assignment.items()))
20         return self.hash_value
21     def __eq__(self, st):
22         return self.assignment == st.assignment
23     def __str__(self):
24         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)_____
26 class Regression_STRIPS(Search_problem):
27     """A search problem where:
28     * a node is a goal to be achieved, represented by a set of propositions.
29     * the dynamics are specified by the STRIPS representation of actions
30     """
31
32     def __init__(self, planning_problem, heur=lambda s,g:0):
33         """creates a regression seach space from a planning problem.
34         heur(state,goal) is a heuristic function;
35         an underestimate of the cost from state to goal, where
36         both state and goals are feature:value dictionaries
37         """
38         self.prob_domain = planning_problem.prob_domain
39         self.top_goal = Subgoal(planning_problem.goal)
40         self.initial_state = planning_problem.initial_state

```

```

41     self.heur = heur
42
43     def is_goal(self, subgoal):
44         """if subgoal is true in the initial state, a path has been found"""
45         goal_asst = subgoal.assignment
46         return all((g in self.initial_state) and (self.initial_state[g]==goal_asst[g])
47                     for g in goal_asst)
48
49     def start_nodes(self):
50         """the list of start nodes consists of the top-level goal"""
51         return [self.top_goal]
52
53     def neighbors(self, subgoal):
54         """returns a list of the arcs for the neighbors of subgoal in this problem"""
55         cost = 1
56         goal_asst = subgoal.assignment
57         return [ Arc(subgoal, self.weakest_precond(act, goal_asst), cost, act)
58                 for act in self.prob_domain.actions
59                 if self.possible(act, goal_asst)]
60
61     def possible(self, act, goal_asst):
62         """True if act is possible to achieve goal_asst.
63
64         the action achieves an element of the effects and
65         the action doesn't delete something that needs to be achieved and
66         the preconditions are consistent with other subgoals that need to be achieved
67         """
68         effects = self.prob_domain.strips_map[act].effects
69         preconds = self.prob_domain.strips_map[act].preconditions
70         return ( any(goal_asst[prop]==effects[prop]
71                     for prop in effects if prop in goal_asst)
72                 and all(goal_asst[prop]==effects[prop]
73                        for prop in effects if prop in goal_asst)
74                 and all(goal_asst[prop]==preconds[prop]
75                        for prop in preconds if prop not in effects and prop in goal_asst)
76                 )
77
78     def weakest_precond(self, act, goal_asst):
79         """returns the subgoal that must be true so goal_asst holds after act"""
80         new_asst = self.prob_domain.strips_map[act].preconditions.copy()
81         for g in goal_asst:
82             if g not in self.prob_domain.strips_map[act].effects:
83                 new_asst[g] = goal_asst[g]
84         return Subgoal(new_asst)
85
86     def heuristic(self, subgoal):
87         """in the regression planner a node is a subgoal.
88         the heuristic is an (under)estimate of the cost of going from the initial state to subgoal.
89         """
90         return self.heur(self.initial_state, subgoal.assignment)

```



```

stripsRegressionPlanner.py — (continued)
92 from searchBranchAndBound import DF_branch_and_bound
93 from searchAStar import Searcher
94 from searchMPP import SearcherMPP
95 from stripsProblem import problem0, problem1, problem2
96
97 # Searcher(Regression_STRIPS(problem0)).search() #A*
98 # SearcherMPP(Regression_STRIPS(problem0)).search() #A* with MPP
99 # DF_branch_and_bound(Regression_STRIPS(problem0),10).search() #B&B

```

**Exercise 6.7** Multiple path pruning could be used to prune more than the current code. In particular, if the current node contains more conditions than a previously visited node, it can be pruned. For example, if  $\{a : \text{True}, b : \text{False}\}$  has been visited, then any node that is a superset, e.g.,  $\{a : \text{True}, b : \text{False}, d : \text{True}\}$ , need not be expanded. If the simpler subgoal does not lead to a solution, the more complicated one won't 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 variables 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) variables is incompatible if no possible (reachable) state can include that assignment. For example,  $\{ 'MW' : \text{True}, 'RHM' : \text{True} \}$  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 should 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)
60 ##### Regression Planner
61 from stripsRegressionPlanner import Regression_STRIPS
62
63 print("\n***** REGRESSION NO HEURISTIC")
64 print(SearcherMPP(Regression_STRIPS(thisproblem)).search())
65

```

```

66 | print("\n***** REGRESSION WITH HEURISTIC")
67 | print(SearcherMPP(Regression_STRIPS(thisproblem,heuristic_fun)).search())

```

**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 better heuristic 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.

Here we implement the CSP planner assuming 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).

This assumes the same action representation as before; we do not consider factored actions (action features), nor do we implement state constraints.

```

_____stripsCSPPlanner.py — CSP planner where actions are represented using STRIPS_____
11 | from cspProblem import CSP, Constraint
12 |
13 | class CSP_from_STRIPS(CSP):
14 |     """A CSP where:
15 |     * a CSP variable is constructed by st(var,stage).
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.act_vars = [st('action',stage) for stage in range(number_stages)]
24 |         domains = {av:prob_domain.actions for av in self.act_vars}
25 |         domains.update({ st(var,stage):dom
26 |                         for (var,dom) in prob_domain.feats_vals.items()
27 |                         for stage in range(number_stages+1)})
28 |         # initial state constraints:
29 |         constraints = [Constraint((st(var,0),), is_(val))
30 |                        for (var,val) in initial_state.items())
31 |         # goal constraints on the final state:
32 |         constraints += [Constraint((st(var,number_stages),),
33 |                                   is_(val))
34 |                        for (var,val) in goal.items())
35 |         # precondition constraints:
36 |         constraints += [Constraint((st(var,stage), st('action',stage)),
37 |                                   if_(val,act)) # st(var,stage)==val if st('action',stage)=act

```

```

38         for act,strips in prob_domain.strips_map.items()
39         for var,val in strips.preconditions.items()
40         for stage in range(number_stages)]
41     # effect constraints:
42     constraints += [Constraint((st(var,stage+1), st('action',stage)),
43                               if_(val,act)) # st(var,stage+1)==val if st('action',stage)==act
44         for act,strips in prob_domain.strips_map.items()
45         for var,val in strips.effects.items()
46         for stage in range(number_stages)]
47     # frame constraints:
48     constraints += [Constraint((st(var,stage), st('action',stage), st(var,stage+1)),
49                               eq_if_not_in_({act for act in prob_domain.actions
50                                                if var in prob_domain.strips_map[act].effects}))
51         for var in prob_domain.feats_vals
52         for stage in range(number_stages) ]
53     CSP.__init__(self, domains, constraints)
54
55     def extract_plan(self,soln):
56         return [soln[a] for a in self.act_vars]
57
58     def st(var,stage):
59         """returns a string for the var-stage pair that can be used as a variable"""
60         return str(var)+"_"+str(stage)

```

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)* turns *True* and *is\_(3)(7)* returns *False*.

Note that the underscore ('\_') is part of the name; here we use it as the convention that it is a function that returns a function. This uses two different styles 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) _____
62 def is_(val):
63     """returns a function that is true when it is it applied to val.
64     """
65     return lambda x: x == val
66
67 def if_(v1,v2):
68     """if the second argument is v2, the first argument must be v1"""
69     #return lambda x1,x2: x1==v1 if x2==v2 else True
70     def if_fun(x1,x2):
71         return x1==v1 if x2==v2 else True
72     if_fun.__doc__ = "if x2 is "+str(v2)+" then x1 is "+str(v1)
73     return if_fun
74
75 def eq_if_not_in_(actset):

```

```

76     """first and third arguments are equal if action is not in actset"""
77     return lambda x1, a, x2: x1==x2 if a not in actset else True

```

Putting it together, this returns a list of actions that solves the problem *prob* 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)
79 def con_plan(prob,horizon):
80     """finds a plan for problem prob given horizon.
81     """
82     csp = CSP_from_STRIPS(prob, horizon)
83     sol = Con_solver(csp).solve_one()
84     return csp.extract_plan(sol) if sol else sol

```

The following are some example queries.

```

stripsCSPPlanner.py — (continued)
86 from searchDepthFirst import Depth_first_search
87 from stripsProblem import delivery_domain
88 from cspConsistency import Search_with_AC_from_CSP, Con_solver
89 from stripsProblem import Planning_problem, problem0, problem1, problem2
90
91 # Problem 0
92 # con_plan(problem0,1) # should it succeed?
93 # con_plan(problem0,2) # should it succeed?
94 # con_plan(problem0,3) # should it succeed?
95 # To use search to enumerate solutions
96 #searcher0a = Depth_first_search(Search_with_AC_from_CSP(CSP_from_STRIPS(problem0, 1)))
97 #print(searcher0a.search())
98
99 ## Problem 1
100 # con_plan(problem1,5) # should it succeed?
101 # con_plan(problem1,4) # should it succeed?
102 ## To use search to enumerate solutions:
103 #searcher15a = Depth_first_search(Search_with_AC_from_CSP(CSP_from_STRIPS(problem1, 5)))
104 #print(searcher15a.search())
105
106 ## Problem 2
107 #con_plan(problem2, 6) # should fail??
108 #con_plan(problem2, 7) # should succeed???
109
110 ## Example 6.13
111 problem3 = Planning_problem(delivery_domain,
112                             {'SWC':True, 'RHC':False}, {'SWC':False})
113 #con_plan(problem3,2) # Horizon of 2
114 #con_plan(problem3,3) # Horizon of 3
115
116 problem4 = Planning_problem(delivery_domain,{'SWC':True},
117                             {'SWC':False, 'MW':False, 'RHM':False})

```

```

118 |
119 | # For the stochastic local search:
120 | #from cspSLS import SLSearcher, Runtime_distribution
121 | # cspplanning15 = CSP_from_STRIPS(problem1, 5) # should succeed
122 | #se0 = SLSearcher(cspplanning15); print(se0.search(100000,0.5))
123 | #p = Runtime_distribution(cspplanning15)
124 | #p.plot_run(1000,1000,0.7) # warning will take a few minutes

```

## 6.5 Partial-Order Planner

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. We need action instances because the same action could be carried out at different times.

```

_____stripsPOP.py — Partial-order Planner using STRIPS representation _____
11 | from searchProblem import Arc, Search_problem
12 | import random
13 |
14 | class Action_instance(object):
15 |     next_index = 0
16 |     def __init__(self,action,index=None):
17 |         if index is None:
18 |             index = Action_instance.next_index
19 |             Action_instance.next_index += 1
20 |             self.action = action
21 |             self.index = index
22 |
23 |     def __str__(self):
24 |         return str(self.action)+"#"+str(self.index)
25 |
26 |     __repr__ = __str__ # __repr__ function is the same as the __str__ function

```

A node (as in the abstraction of search space) in a partial-order planner 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. Here we represent the set of pairs that are in transitive closure of the *before* relation. This lets us quickly determine whether some before relation is consistent with the current constraints.

- *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  $(a_0, g, a_1)$  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)
28 class POP_node(object):
29     """a (partial) partial-order plan. This is a node in the search space."""
30     def __init__(self, actions, constraints, agenda, causal_links):
31         """
32         * actions is a set of action instances
33         * constraints a set of (a0,a1) pairs, representing a0<a1,
34           closed under transitivity
35         * agenda list of (subgoal,action) pairs to be achieved, where
36           subgoal is a (variable,value) pair
37         * causal_links is a set of (a0,g,a1) triples,
38           where ai are action instances, and g is a (variable,value) pair
39         """
40         self.actions = actions # a set of action instances
41         self.constraints = constraints # a set of (a0,a1) pairs
42         self.agenda = agenda # list of (subgoal,action) pairs to be achieved
43         self.causal_links = causal_links # set of (a0,g,a1) triples
44
45     def __str__(self):
46         return ("actions: "+str({str(a) for a in self.actions})+
47             "\nconstraints: "+
48             str({(str(a1),str(a2)) for (a1,a2) in self.constraints})+
49             "\nagenda: "+
50             str([(str(s),str(a)) for (s,a) in self.agenda])+
51             "\ncausal_links:"+
52             str({(str(a0),str(g),str(a2)) for (a0,g,a2) in self.causal_links})
53         )

```

*extract\_plan* constructs a total order of action instances that is consistent with the partial order.

```

stripsPOP.py — (continued)
54 def extract_plan(self):
55     """returns a total ordering of the action instances consistent
56     with the constraints.
57     raises IndexError if there is no choice.
58     """
59     sorted_acts = []
60     other_acts = set(self.actions)
61     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, we need to define the start nodes, the goal, and the neighbors of a node.

```

_____stripsPOP.py — (continued)_____
68 from utilities 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_nodes(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 is a coroutine that enumerates the neighbors of a given node.

```

_____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,
101                               POP_node(node.actions, consts2, new_agenda, new_cls),
102                               cost=0)
103         for a0 in self.planning_problem.prob_domain.strips_map: #a0 is an action
104             if self.achieves(a0, subgoal):
105                 #a0 achieves subgoal
106                 new_a = Action_instance(a0)

```

```

107         self.display(2, " using new action", new_a)
108         new_actions = node.actions + [new_a]
109         consts1 = self.add_constraint((self.start, new_a), node.constraints)
110         consts2 = self.add_constraint((new_a, act1), consts1)
111         preconds = self.planning_problem.prob_domain.strips_map[a0].preconditions
112         new_agenda = new_agenda + [(pre, new_a) for pre in preconds.items()]
113         new_clink = (new_a, subgoal, act1)
114         new_cls = node.causal_links + [new_clink]
115         for consts3 in self.protect_all_cls(node.causal_links, new_a, consts2):
116             for consts4 in self.protect_cl_for_actions(node.actions, consts3, new_clink):
117                 yield Arc(node,
118                           POP_node(new_actions, consts4, new_agenda, new_cls),
119                           cost=1)

```

Given a casual 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)
121 def protect_cl_for_actions(self, actions, constrs, clink):
122     """yields constraints that extend constrs and
123     protect causal link (a0, subgoal, a1)
124     for each action in actions
125     """
126     if actions:
127         a=actions[0]
128         rem_actions = actions[1:]
129         a0, subgoal, a1 = clink
130         if a != a0 and a != a1 and self.deletes(a, subgoal):
131             if self.possible((a, a0), constrs):
132                 new_const = self.add_constraint((a, a0), constrs)
133                 yield from self.protect_cl_for_actions(rem_actions, new_const, clink)
134             if self.possible((a1, a), constrs):
135                 new_const = self.add_constraint((a1, a), constrs)
136                 yield from self.protect_cl_for_actions(rem_actions, new_const, clink)
137         else:
138             yield from self.protect_cl_for_actions(rem_actions, constrs, clink)
139     else:
140         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)
142 def protect_all_cls(self, clinks, act, constrs):
143     """yields constraints that protect all causal links from act"""
144     if clinks:
145         (a0, cond, a1) = clinks[0] # select a causal link

```



```

146         rem_clinks = clinks[1:] # remaining causal links
147         if act != a0 and act != a1 and self.deletes(act,cond):
148             if self.possible((act,a0),constrs):
149                 new_const = self.add_constraint((act,a0),constrs)
150                 yield from self.protect_all_cls(rem_clinks,act,new_const)
151             if self.possible((a1,act),constrs):
152                 new_const = self.add_constraint((a1,act),constrs)
153                 yield from self.protect_all_cls(rem_clinks,act,new_const)
154         else:
155             yield from self.protect_all_cls(rem_clinks,act,constrs)
156     else:
157         yield constrs

```

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

```

stripsPOP.py — (continued)
159 def achieves(self,action,subgoal):
160     var,val = subgoal
161     return var in self.effects(action) and self.effects(action)[var] == val
162
163 def deletes(self,action,subgoal):
164     var,val = subgoal
165     return var in self.effects(action) and self.effects(action)[var] != val
166
167 def effects(self,action):
168     """returns the variable:value dictionary of the effects of action.
169     works for both actions and action instances"""
170     if isinstance(action, Action_instance):
171         action = action.action
172     if action == "start":
173         return self.planning_problem.initial_state
174     elif action == "finish":
175         return {}
176     else:
177         return self.planning_problem.prob_domain.strips_map[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)
179 def add_constraint(self, pair, const):
180     if pair in const:
181         return const
182     todo = [pair]
183     newconst = const.copy()
184     while todo:
185         x0,x1 = todo.pop()
186         newconst.add((x0,x1))

```

```

187         for x,y in newconst:
188             if x==x1 and (x0,y) not in newconst:
189                 todo.append((x0,y))
190             if y==x0 and (x,x1) not in newconst:
191                 todo.append((x,x1))
192         return newconst
193
194     def possible(self,pair,constraint):
195         (x,y) = pair
196         return (y,x) not in constraint

```

Some code for testing:

---

```

stripsPOP.py — (continued)
198 from searchBranchAndBound import DF_branch_and_bound
199 from searchAStar import Searcher
200 from searchMPP import SearcherMPP
201 from stripsProblem import problem0, problem1, problem2
202
203 rplanning0 = POP_search_from_STRIPS(problem0)
204 rplanning1 = POP_search_from_STRIPS(problem1)
205 rplanning2 = POP_search_from_STRIPS(problem2)
206 searcher0 = DF_branch_and_bound(rplanning0,5)
207 searcher0a = Searcher(rplanning0)
208 searcher1 = DF_branch_and_bound(rplanning1,10)
209 searcher1a = Searcher(rplanning1)
210 searcher2 = DF_branch_and_bound(rplanning2,10)
211 searcher2a = Searcher(rplanning2)
212 # Try one of the following searchers
213 # a = searcher0.search()
214 # a = searcher0a.search()
215 # a.end().extract_plan() # print a plan found
216 # a.end().constraints # print the constraints
217 # Searcher.max_display_level = 0 # less detailed display
218 # DF_branch_and_bound.max_display_level = 0 # less detailed display
219 # a = searcher1.search()
220 # a = searcher1a.search()
221 # a = searcher2.search()
222 # a = searcher2a.search()

```

# Chapter 7

---

## Learning

A good source of datasets is the UCI machine Learning Repository [?]; the SPECT and car datasets are from this repository.

### 7.1 Representations of Data and Predictions

A **data set** is an enumeration of examples. Each **example** is a list (or tuple) of property values. The property values can be numbers or strings. A **feature** is a function from the examples into the range of the feature. We assume each feature has a variable frange that gives the range of the feature. A **Boolean feature** is a function from the examples into  $\{False, True\}$ . So  $f(e)$  is either *True* or *False*, where  $f$  is a feature and  $e$  is an example. The `__doc__` variable contains the docstring, a string description of the function.

```
learnProblem.py — A Learning Problem
11 import math, random
12 import csv
13 from utilities import Displayable
14
15 boolean = [False, True]
```

When creating a data set, we partition the data into a training set (*train*) and a test set (*test*). The target property is the property that we are making a prediction of.

```
learnProblem.py — (continued)
17 class Data_set(Displayable):
18     """ A data set consists of a list of training data and a list of test data.
19     """
20     seed = None #123456 # make it None for a different test set each time
21
```

```

22     def __init__(self, train, test=None, prob_test=0.30, target_index=0, header=None):
23         """A dataset for learning.
24         train is a list of tuples representing the training examples
25         test is the list of tuples representing the test examples
26         if test is None, a test set is created by selecting each
27             example with probability prob_test
28         target_index is the index of the target. If negative, it counts from right.
29             If target_index is larger than the number of properties,
30             there is no target (for unsupervised learning)
31         header is a list of names for the features
32         """
33         if test is None:
34             train, test = partition_data(train, prob_test, seed=self.seed)
35         self.train = train
36         self.test = test
37         self.display(1, "Tuples read. \nTraining set", len(train),
38                     "examples. Number of columns:", {len(e) for e in train},
39                     "\nTest set", len(test),
40                     "examples. Number of columns:", {len(e) for e in test}
41                     )
42         self.prob_test = prob_test
43         self.num_properties = len(self.train[0])
44         if target_index < 0: #allows for -1, -2, etc.
45             target_index = self.num_properties + target_index
46         self.target_index = target_index
47         self.header = header
48         self.create_features()
49         self.display(1, "There are", len(self.input_features), "features")

```

Initially we assume that all of the properties can be mapped directly into features. If all values are 0 or 1 they can be used as Boolean features. This will be overridden to allow for more general features.

```

learnProblem.py — (continued)
51     def create_features(self):
52         """create the input features and target feature.
53         This assumes that the features all have domain {0,1}.
54         This should be overridden if the features have a different domain.
55         """
56         self.input_features = []
57         for i in range(self.num_properties):
58             def feat(e, index=i):
59                 return e[index]
60             if self.header:
61                 feat.__doc__ = self.header[i]
62             else:
63                 feat.__doc__ = "e["+str(i)+"]"
64             feat.frange = [0,1]
65             if i == self.target_index:
66                 self.target = feat
67             else:

```

```
68 | self.input_features.append(feat)
```

### 7.1.1 Evaluating Predictions

A **predictor** is a function that takes an example and makes a prediction on the value of the target feature. A predictor can be judged according to a number of evaluation criteria. The function *evaluate\_dataset* returns the average error for each example, where the error for each example depends on the evaluation criteria. Here we consider three evaluation criteria, the sum-of-squares, the sum of absolute errors and the logloss (the negative log-likelihood, which is the number of bits to describe the data using a code based on the prediction treated as a probability).

```

learnProblem.py — (continued)
70 | evaluation_criteria = ["sum_squares", "sum_absolute", "logloss"]
71 |
72 | def evaluate_dataset(self, data, predictor, evaluation_criterion):
73 |     """Evaluates predictor on data according to the evaluation_criterion.
74 |     predictor is a function that takes an example and returns a
75 |     prediction for the target feature.
76 |     evaluation_criterion is one of the evaluation_criteria.
77 |     """
78 |     assert evaluation_criterion in self.evaluation_criteria, "given: "+str(evaluation_criterion)
79 |     if data:
80 |         try:
81 |             error = sum(error_example(predictor(example), self.target(example),
82 |                                     evaluation_criterion)
83 |                         for example in data)/len(data)
84 |         except ValueError:
85 |             return float("inf") # infinity
86 |         return error

```

*error\_example* is used to evaluate a single example, based on the predicted value, the actual value and the evaluation criterion. Note that for logloss, the actual value must be 0 or 1.

```

learnProblem.py — (continued)
88 | def error_example(predicted, actual, evaluation_criterion):
89 |     """returns the error of the for the predicted value given the actual value
90 |     according to evaluation_criterion.
91 |     Throws ValueError if the error is infinite (log(0))
92 |     """
93 |     if evaluation_criterion=="sum_squares":
94 |         return (predicted-actual)**2
95 |     elif evaluation_criterion=="sum_absolute":
96 |         return abs(predicted-actual)
97 |     elif evaluation_criterion=="logloss":
98 |         assert actual in [0,1], "actual="+str(actual)
99 |         if actual==0:

```

```

100         return -math.log2(1-predicted)
101     else:
102         return -math.log2(predicted)
103     elif evaluation_criterion=="characteristic_ss":
104         return sum((1-predicted[i])**2 if actual==i else predicted[i]**2
105                    for i in range(len(predicted)))
106     else:
107         raise RuntimeError("Not evaluation criteria: "+str(evaluation_criterion))

```

### 7.1.2 Creating Test and Training Sets

The following method partitions the data into a training set and a test set. 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 data set, which we 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)
109 def partition_data(data, prob_test=0.30, seed=None):
110     """partitions the data into a training set and a test set, where
111     prob_test is the probability of each example being in the test set.
112     """
113     train = []
114     test = []
115     if seed:    # given seed makes the partition consistent from run-to-run
116         random.seed(seed)
117     for example in data:
118         if random.random() < prob_test:
119             test.append(example)
120         else:
121             train.append(example)
122     return train, test

```

### 7.1.3 Importing Data From File

A data set is typically loaded from a file. The default here is that it loaded from a CSV (comma separated values) file, although the default separator can be changed. This assumes that all lines that contain the separator are valid data (so we only include 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\_aa* and commenting out the following line. *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.

This allows for some of the columns to be included. Note that if *include\_only* is specified, the target index is in the resulting

```

learnProblem.py — (continued)
124 class Data_from_file(Data_set):
125     def __init__(self, file_name, separator=',', num_train=None, prob_test=0.3,
126                 has_header=False, target_index=0, boolean_features=True,
127                 categorical=[], include_only=None):
128         """create a dataset from a file
129         separator is the character that separates the attributes
130         num_train is a number n specifying the first n tuples are training, or None
131         prob_test is the probability an example should be in the test set (if num_train is None)
132         has_header is True if the first line of file is a header
133         target_index specifies which feature is the target
134         boolean_features specifies whether we want to create Boolean features
135         (if False, is uses the original features).
136         categorical is a set (or list) of features that should be treated as categorical
137         include_only is a list or set of indexes of columns to include
138         """
139         self.boolean_features = boolean_features
140         with open(file_name, 'r', newline='') as csvfile:
141             # data_all = csv.reader(csvfile, delimiter=separator) # for more complicated CSV files
142             data_all = (line.strip().split(separator) for line in csvfile)
143             if include_only is not None:
144                 data_all = [(v for (i, v) in enumerate(line) if i in include_only) for line in data_all]
145             if has_header:
146                 header = next(data_all)
147             else:
148                 header = None
149             data_tuples = (make_num(d) for d in data_all if len(d) > 1)
150             if num_train is not None:
151                 # training set is divided into training then test examples
152                 # the file is only read once, and the data is placed in appropriate list
153                 train = []
154                 for i in range(num_train): # will give an error if insufficient examples
155                     train.append(next(data_tuples))
156                 test = list(data_tuples)
157                 Data_set.__init__(self, train, test=test, target_index=target_index, header=header)
158             else: # randomly assign training and test examples
159                 Data_set.__init__(self, data_tuples, prob_test=prob_test,
160                                 target_index=target_index, header=header)
161
162     def __str__(self):
163         if self.train and len(self.train) > 0:
164             return ("Data: " + str(len(self.train)) + " training examples, ")

```

```

165         +str(len(self.test))+ " test examples, "
166         +str(len(self.train[0]))+" features.")
167     else:
168         return ("Data: "+str(len(self.train))+ " training examples, "
169               +str(len(self.test))+ " test examples.")

```

### 7.1.4 Creating Binary Features

Some of the algorithms require Boolean features or features with domain  $\{0, 1\}$ . In order to be able to use these on datasets that allow for arbitrary ranges of input variables, we construct binary features from the attributes. This method overrides the method in *Data\_set*.

There are 3 cases:

- When the attribute only has two values, we designate one to be the “true” value.
- When the values are all numeric, we assume they are ordered (as opposed to just being some classes that happen to be labelled with numbers, but where the numbers have no meaning) and construct Boolean features for splits of the data. That is, the feature is  $e[ind] < cut$  for some value *cut*. We choose a number of *cut* values, up to a maximum number of cuts, given by *max\_num\_cuts*.
- When the values are not all numeric, we assume they are unordered, and create 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 set because we haven’t seen the test set. For the examples in the test set with that value, the indicator functions return false.

```

learnProblem.py — (continued)
171 def create_features(self, max_num_cuts=8):
172     """creates boolean features from input features.
173     max_num_cuts is the maximum number of binary variables
174     to split a numerical feature into.
175     """
176     ranges = [set() for i in range(self.num_properties)]
177     for example in self.train:
178         for ind, val in enumerate(example):
179             ranges[ind].add(val)
180     if self.target_index <= self.num_properties:
181         def target(e, index=self.target_index):
182             return e[index]
183         if self.header:
184             target.__doc__ = self.header[ind]

```



```

185         else:
186             target.__doc__ = "e["+str(ind)+"]"
187             target.frange = ranges[self.target_index]
188             self.target = target
189     if self.boolean_features:
190         self.input_features = []
191         for ind,frange in enumerate(ranges):
192             if ind != self.target_index and len(frange)>1:
193                 if len(frange) == 2:
194                     # two values, the feature is equality to one of them.
195                     true_val = list(frange)[1] # choose one as true
196                     def feat(e, i=ind, tv=true_val):
197                         return e[i]==tv
198                     if self.header:
199                         feat.__doc__ = self.header[ind]+"==" +str(true_val)
200                     else:
201                         feat.__doc__ = "e["+str(ind)+"]==" +str(true_val)
202                     feat.frange = boolean
203                     self.input_features.append(feat)
204                 elif all(isinstance(val,(int,float)) for val in frange):
205                     # all numeric, create cuts of the data
206                     sorted_frange = sorted(frange)
207                     num_cuts = min(max_num_cuts,len(frange))
208                     cut_positions = [len(frange)*i//num_cuts for i in range(1,num_cuts)]
209                     for cut in cut_positions:
210                         cutat = sorted_frange[cut]
211                         def feat(e, ind=ind, cutat=cutat):
212                             return e[ind_] < cutat
213
214                     if self.header:
215                         feat.__doc__ = self.header[ind]+"<" +str(cutat)
216                     else:
217                         feat.__doc__ = "e["+str(ind)+"]<" +str(cutat)
218                     feat.frange = boolean
219                     self.input_features.append(feat)
220                 else:
221                     # create an indicator function for every value
222                     for val in frange:
223                         def feat(e, ind=ind, val=val):
224                             return e[ind_] == val_
225                         if self.header:
226                             feat.__doc__ = self.header[ind]+"==" +str(val)
227                         else:
228                             feat.__doc__ = "e["+str(ind)+"]==" +str(val)
229                         feat.frange = boolean
230                         self.input_features.append(feat)
231     else: # boolean_features is off
232         self.input_features = []
233         for i in range(self.num_properties):
234             def feat(e,index=i):

```

```

235         return e[index]
236     if self.header:
237         feat.__doc__ = self.header[i]
238     else:
239         feat.__doc__ = "e["+str(i)+"]"
240     feat.frange = ranges[i]
241     if i == self.target_index:
242         self.target = feat
243     else:
244         self.input_features.append(feat)

```

**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, and if there are 30 elements (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 ranges is 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 data sets).

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

```

learnProblem.py — (continued)
245 def make_num(str_list):
246     """make the elements of string list str_list numerical if possible.
247     Otherwise remove initial and trailing spaces.
248     """
249     res = []
250     for e in str_list:
251         try:
252             res.append(int(e))
253         except ValueError:
254             try:
255                 res.append(float(e))
256             except ValueError:
257                 res.append(e.strip())
258     return res

```

### 7.1.5 Augmented Features

Sometimes we want to augment the features with new features computed from the old features (eg. the product of features). Here we allow the creation of a new dataset from an old dataset but with new features.

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)
260 class Data_set_augmented(Data_set):
261     def __init__(self, dataset, unary_functions=[], binary_functions=[], include_orig=True):
262         """creates a dataset like dataset but with new features
263         unary_function is a list of unary feature constructors
264         binary_functions is a list of binary feature combiners.
265         include_orig specifies whether the original features should be included
266         """
267         self.orig_dataset = dataset
268         self.unary_functions = unary_functions
269         self.binary_functions = binary_functions
270         self.include_orig = include_orig
271         self.target = dataset.target
272         Data_set.__init__(self, dataset.train, test=dataset.test,
273                           target_index = dataset.target_index)
274
275     def create_features(self):
276         if self.include_orig:
277             self.input_features = self.orig_dataset.input_features.copy()
278         else:
279             self.input_features = []
280         for u in self.unary_functions:
281             for f in self.orig_dataset.input_features:
282                 self.input_features.append(u(f))
283         for b in self.binary_functions:
284             for f1 in self.orig_dataset.input_features:
285                 for f2 in self.orig_dataset.input_features:
286                     if f1 != f2:
287                         self.input_features.append(b(f1,f2))

```

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

```

learnProblem.py — (continued)
289 def square(f):
290     """a unary feature constructor to construct the square of a feature
291     """
292     def sq(e):
293         return f(e)**2
294     sq.__doc__ = f.__doc__+"**2"
295     return sq
296
297 def power_feat(n):
298     """given n returns a unary feature constructor to construct the nth power of a feature.
299     e.g., power_feat(2) is the same as square
300     """
301     def fn(f,n=n):

```

```

302     def pow(e,n=n):
303         return f(e)**n
304     pow.__doc__ = f.__doc__+"**"+str(n)
305     return pow
306 return fn
307
308 def prod_feat(f1,f2):
309     """a new feature that is the product of features f1 and f2
310     """
311     def feat(e):
312         return f1(e)*f2(e)
313     feat.__doc__ = f1.__doc__+"*"+f2.__doc__
314     return feat
315
316 def eq_feat(f1,f2):
317     """a new feature that is 1 if f1 and f2 give same value
318     """
319     def feat(e):
320         return 1 if f1(e)==f2(e) else 0
321     feat.__doc__ = f1.__doc__+"==" +f2.__doc__
322     return feat
323
324 def xor_feat(f1,f2):
325     """a new feature that is 1 if f1 and f2 give different values
326     """
327     def feat(e):
328         return 1 if f1(e)!=f2(e) else 0
329     feat.__doc__ = f1.__doc__+"!=" +f2.__doc__
330     return feat

```

Example:

```

learnProblem.py — (continued)
332 # from learnProblem import Data_set_augmented,prod_feat
333 # data = Data_from_file('data/holiday.csv', num_train=19, target_index=-1)
334 ## data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0)
335 # dataplus = Data_set_augmented(data,[],[prod_feat])
336 # dataplus = Data_set_augmented(data,[],[prod_feat,xor_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 property with them). Change *construct\_features* so that it does not create both versions for symmetric combiners.

### 7.1.6 Learner

A learner takes a dataset (and possible other arguments specific to the method). To get it to learn, we call the *learn()* method. This implements *Displayable* so

that we can display traces at multiple levels of detail (and perhaps with a GUI).

```

337 from utilities import Displayable
338
339 class Learner(Displayable):
340     def __init__(self, dataset):
341         raise NotImplementedError("Learner.__init__") # abstract method
342
343     def learn(self):
344         """returns a predictor, a function from a tuple to a value for the target feature
345         """
346         raise NotImplementedError("learn") # abstract method

```

## 7.2 Learning With No Input Features

If we make the same prediction for each example, what prediction should we make?

There are a few alternatives as to what could be allowed in a prediction:

- a point prediction, where we are only allowed to predict one of the values of the feature. For example, if the values of the feature are  $\{0, 1\}$  we are only allowed to predict 0 or 1 or if the values are ratings in  $\{1, 2, 3, 4, 5\}$ , we can only predict one of these integers.
- a point prediction, where we are allowed to predict any value. For example, if the values of the feature are  $\{0, 1\}$  we may be allowed to predict 0.3, 1, or even 1.7. For all of the criteria we can imagine, there is no point in predicting a value greater than 1 or less than zero (but that doesn't mean we can't), but it is often useful to predict a value between 0 and 1. If the values are ratings in  $\{1, 2, 3, 4, 5\}$ , we may want to predict 3.4.
- a probability distribution over the values of the feature. For each value  $v$ , we predict a non-negative number  $p_v$ , such that the sum over all predictions is 1.

The following code assumes the second of these, where we can make a point prediction of any value (although median will only predict one of the actual values for the feature).

The *point\_prediction* function takes in a target feature (which is assumed to be numeric), some training data, and a section of what to return, and returns a function that takes in an example, and makes a prediction of a value for the target variable, but makes same prediction for all examples.

This method uses *selection*, whose value should be "median", "proportion", or "Laplace" determine what prediction should be made.

learnNoInputs.py — Learning ignoring all input features

```

11 from learnProblem import Learner, Data_set
12 import math, random
13
14 selections = ["median", "mean", "Laplace"]
15
16 def point_prediction(target, training_data,
17                     selection="mean" ):
18     """makes a point prediction for a set of training data.
19     target provides the target
20     training_data provides the training data to use (often a subset of train).
21     selection specifies what statistic of the data to use as the evaluation.
22     to_optimize provides a criteria to optimize (used to guess selection)
23     """
24     assert len(training_data)>0
25     if selection == "median":
26         counts,total = target_counts(target,training_data)
27         middle = total/2
28         cumulative = 0
29         for val,num in sorted(counts.items()):
30             cumulative += num
31             if cumulative > middle:
32                 break # exit loop with val as the median
33     elif selection == "mean":
34         val = mean((target(e) for e in training_data))
35     elif selection == "Laplace":
36         val = mean((target(e) for e in training_data),len(target.frange),1)
37     elif selection == "mode":
38         raise NotImplementedError("mode")
39     else:
40         raise RuntimeError("Not valid selection: "+str(selection))
41     fun = lambda x: val
42     fun.__doc__ = str(val)
43     return fun
44
45 def mean(enum,count=0,sum=0):
46     """returns the mean of enumeration enum,
47     count and sum are initial counts and the initial sum.
48     This works for enumerations, even where len() is not defined"""
49     for e in enum:
50         count += 1
51         sum += e
52     return sum/count
53
54 def target_counts(target, data_subset):
55     """returns a value:count dictionary of the count of the number of
56     times target has this value in data_subset, and the number of examples.
57     """
58     counts = {val:0 for val in target.frange}
59     total = 0

```

```

60     for instance in data_subset:
61         total += 1
62         counts[target(instance)] += 1
63     return counts, total

```

### 7.2.1 Testing

To test the point prediction, we will first generate some data from a simple (Bernoulli) distribution, where there are two possible values, 0 and 1 for the target property. Given *prob*, a number in the range  $[0, 1]$ , this generate some training and test data where *prob* is the probability of each example being 1.

```

learnNoInputs.py — (continued)
65 class Data_set_random(Data_set):
66     """A data set of a {0,1} feature generated randomly given a probability"""
67     def __init__(self, prob, train_size, test_size=100):
68         """a data set of with train_size training examples,
69         test_size test examples
70         where each examples in generated where prob i the probability of 1
71         """
72         train = [[1] if random.random()<prob else [0] for i in range(train_size)]
73         test = [[1] if random.random()<prob else [0] for i in range(test_size)]
74         Data_set.__init__(self, train, test, target_index=0)

```

Let's try to evaluate the predictions of the possible selections according to the different evaluation criteria, for various training sizes.

```

learnNoInputs.py — (continued)
76 def test_no_inputs():
77     num_samples = 1000 #number of runs to average over
78     test_size = 100 # number of test examples for each prediction
79     for train_size in [1,2,3,4,5,10,20,100,1000]:
80         total_error = {(select,crit):0
81                         for select in selections
82                         for crit in Data_set.evaluation_criteria}
83     for sample in range(num_samples): # average over num_samples
84         p = random.random()
85         data = Data_set_random(p, train_size, test_size)
86         for select in selections:
87             prediction = point_prediction(data.target, data.train, selection=select)
88             for ecrit in Data_set.evaluation_criteria:
89                 test_error = data.evaluate_dataset(data.test,prediction,ecrit)
90                 total_error[(select,ecrit)] += test_error
91     print("For training size",train_size,":")
92     for ecrit in Data_set.evaluation_criteria:
93         print("    Evaluated according to",ecrit,":")
94         for select in selections:
95             print("        Average error of",select,"is",
96                   total_error[(select,ecrit)]/num_samples)
97

```

```

98 | if __name__ == "__main__":
99 |     test_no_inputs()

```

## 7.3 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. It stops splitting if there are no input features, the number of examples is less than a specified number of examples or all of the examples agree on the target property.

```

_____learnDT.py — Learning a binary decision tree_____
11 | from learnProblem import Learner, error_example
12 | from learnNoInputs import point_prediction, target_counts, selections
13 | import math
14 |
15 | class DT_learner(Learner):
16 |     def __init__(self,
17 |                 dataset,
18 |                 to_optimize="sum_squares",
19 |                 leaf_selection="mean", # what to use for point prediction at leaves
20 |                 train=None,           # used for cross validation
21 |                 min_number_examples=10):
22 |         self.dataset = dataset
23 |         self.target = dataset.target
24 |         self.to_optimize = to_optimize
25 |         self.leaf_selection = leaf_selection
26 |         self.min_number_examples = min_number_examples
27 |         if train is None:
28 |             self.train = self.dataset.train
29 |         else:
30 |             self.train = train
31 |
32 |     def learn(self):
33 |         return self.learn_tree(self.dataset.input_features, self.train)

```

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 doesn't split if there are no more input features, if there are fewer examples than *min\_number\_examples*, if all the examples agree on the value of the target or if the best split makes all examples in the same partition



If it decides to split, it selects the best split and returns the condition to split on (in the variable *split*) and the corresponding partition of the examples.

```

learnDT.py — (continued)
35 def learn_tree(self, input_features, data_subset):
36     """returns a decision tree
37     for input_features is a set of possible conditions
38     data_subset is a subset of the data used to build this (sub)tree
39
40     where a decision tree is a function that takes an example and
41     makes a prediction on the target feature
42     """
43     if (input_features and len(data_subset) >= self.min_number_examples):
44         first_target_val = self.target(data_subset[0])
45         allagree = all(self.target(inst)==first_target_val for inst in data_subset)
46         if not allagree:
47             split, partn = self.select_split(input_features, data_subset)
48             if split: # the split succeeded in splitting the data
49                 false_examples, true_examples = partn
50                 rem_features = [fe for fe in input_features if fe != split]
51                 self.display(2, "Splitting on", split.__doc__, "with examples split",
52                             len(true_examples), ":", len(false_examples))
53                 true_tree = self.learn_tree(rem_features, true_examples)
54                 false_tree = self.learn_tree(rem_features, false_examples)
55                 def fun(e):
56                     if split(e):
57                         return true_tree(e)
58                     else:
59                         return false_tree(e)
60                 #fun = lambda e: true_tree(e) if split(e) else false_tree(e)
61                 fun.__doc__ = ("if "+split.__doc__+" then (" + true_tree.__doc__ +
62                             ") else (" + false_tree.__doc__ + ")")
63                 return fun
64             # don't expand the trees but return a point prediction
65             return point_prediction(self.target, data_subset, selection=self.leaf_selection)

learnDT.py — (continued)
67 def select_split(self, input_features, data_subset):
68     """finds best feature to split on.
69
70     input_features is a non-empty list of features.
71     returns feature, partition
72     where feature is an input feature with the smallest error as
73     judged by to_optimize or
74     feature==None if there are no splits that improve the error
75     partition is a pair (false_examples, true_examples) if feature is not None
76     """
77     best_feat = None # best feature
78     # best_error = float("inf") # infinity - more than any error
79     best_error = training_error(self.dataset, data_subset, self.to_optimize)

```

```

80     best_partition = None
81     for feat in input_features:
82         false_examples, true_examples = partition(data_subset, feat)
83         if false_examples and true_examples: #both partitons are non-empty
84             err = (training_error(self.dataset, false_examples, self.to_optimize)
85                   + training_error(self.dataset, true_examples, self.to_optimize))
86             self.display(3, " split on", feat.__doc__, "has err=", err,
87                          "splits into", len(true_examples), ":", len(false_examples))
88             if err < best_error:
89                 best_feat = feat
90                 best_error = err
91                 best_partition = false_examples, true_examples
92     self.display(3, "best split is on", best_feat.__doc__,
93                "with err=", best_error)
94     return best_feat, best_partition
95
96 def partition(data_subset, feature):
97     """partitions the data_subset by the feature"""
98     true_examples = []
99     false_examples = []
100    for example in data_subset:
101        if feature(example):
102            true_examples.append(example)
103        else:
104            false_examples.append(example)
105    return false_examples, true_examples
106
107
108 def training_error(dataset, data_subset, to_optimize):
109     """returns training error for dataset on to_optimize.
110     This assumes that we choose the best value for the optimization
111     criteria for dataset according to point_prediction
112     """
113     select_dict = {"sum_squares": "mean", "sum_absolute": "median",
114                  "logloss": "Laplace"} # arbitrary mapping. Perhaps wrong.
115     selection = select_dict[to_optimize]
116     predictor = point_prediction(dataset.target, data_subset, selection=selection)
117     error = sum(error_example(predictor(example),
118                             dataset.target(example),
119                             to_optimize)
120                for example in data_subset)
121     return error

```

Test cases:

---

```

123 from learnProblem import Data_set, Data_from_file
124
125 def test(data):
126     """Prints errors and the trees for various evaluation criteria and ways to select leaves.
127     """

```

```

128     for crit in Data_set.evaluation_criteria:
129         for leaf in selections:
130             tree = DT_learner(data, to_optimize=crit, leaf_selection=leaf).learn()
131             print("For",crit,"using",leaf,"at leaves, tree built is:",tree.__doc__)
132             if data.test:
133                 for ecrit in Data_set.evaluation_criteria:
134                     test_error = data.evaluate_dataset(data.test, tree, ecrit)
135                     print("    Average error for", ecrit,"using",leaf, "at leaves is", test_error)
136
137 if __name__ == "__main__":
138     #print("carbool.csv"); test(data = Data_from_file('data/carbool.csv', target_index=-1))
139     # print("SPECT.csv"); test(data = Data_from_file('data/SPECT.csv', target_index=0))
140     print("mail_reading.csv"); test(data = Data_from_file('data/mail_reading.csv', target_index=-1))
141     # print("holiday.csv"); test(data = Data_from_file('data/holiday.csv', num_train=19, target_in

```

**Exercise 7.4** 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 7.5** 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 than 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 7.6** 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 7.7** Some people have suggested using information gain (which is equivalent to greedy optimization of logloss) 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?

## 7.4 Cross Validation and Parameter Tuning

To run the cross validation demo, in folder "aipython", load "learnCrossValidation.py", using e.g., `ipython -i learnCrossValidation.py`. Run `plot_fig_7.15()` to produce a graph like Figure 7.15. Note that different runs will produce different graphs, so your graph will not look like the one in the textbook. To try more examples, copy and paste the commented-out commands at the bottom of that file. This requires Python 3 with matplotlib.

The above decision tree overfits 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 data set, we can only use the training data (and not the test data) to tune the parameter.

In  $k$ -fold cross validation, we partition the training set into  $k$  approximately equal-sized folds (each fold is an enumeration of examples). For each fold, we train on the other examples, and determine the error of the prediction on that fold. For example, if there are 10 folds, we train on 90% of the data, and then test on remaining 10% of the data. We do 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
11 from learnProblem import Data_set, Data_from_file, error_example
12 from learnDT import DT_learner
13 import matplotlib.pyplot as plt
14 import random
15
16 class K_fold_dataset(object):
17     def __init__(self, training_set, num_folds):
18         self.data = training_set.train.copy()
19         self.target = training_set.target
20         self.input_features = training_set.input_features
21         self.num_folds = num_folds
22         random.shuffle(self.data)
23         self.fold_boundaries = [(len(self.data)*i)//num_folds
24                                for i in range(0,num_folds+1)]
25
26     def fold(self, fold_num):
27         for i in range(self.fold_boundaries[fold_num],
28                        self.fold_boundaries[fold_num+1]):
29             yield self.data[i]
30

```

```

31     def fold_complement(self, fold_num):
32         for i in range(0, self.fold_boundaries[fold_num]):
33             yield self.data[i]
34         for i in range(self.fold_boundaries[fold_num+1], len(self.data)):
35             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)
37     def validation_error(self, learner, criterion, **other_params):
38         error = 0
39         try:
40             for i in range(self.num_folds):
41                 predictor = learner(self, train=list(self.fold_complement(i)),
42                                     **other_params).learn()
43                 error += sum( error_example(predictor(example),
44                                     self.target(example),
45                                     criterion)
46                             for example in self.fold(i))
47         except ValueError:
48             return float("inf") #infinity
49         return error/len(self.data)

```

The `plot_error` method plots the average error as a function of a 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 then it cannot be used to test.

```

learnCrossValidation.py — (continued)
51     def plot_error(data, criterion="sum_squares", num_folds=5, xscale='log'):
52         """Plots the error on the validation set and the test set
53         with respect to settings of the minimum number of examples.
54         xscale should be 'log' or 'linear'
55         """
56         plt.ion()
57         plt.xscale('linear') # change between log and linear scale
58         plt.xlabel("min_number_examples")
59         plt.ylabel(criterion+" error")
60         folded_data = K_fold_dataset(data, num_folds)
61         verrors = [] # validation errors
62         terrors = [] # test set errors
63         for mne in range(1, len(data.train)+2):
64             verrors.append(folded_data.validation_error(DT_learner, criterion,
65                                                         min_number_examples=mne))
66             tree = DT_learner(data, criterion, min_number_examples=mne).learn()
67             terrors.append(data.evaluate_dataset(data.test, tree, criterion))
68         plt.plot(range(1, len(data.train)+2), verrors, ls='-', color='k', label="validation for "+criterion)
69         plt.plot(range(1, len(data.train)+2), terrors, ls='--', color='k', label="test set for "+criterion)

```



*predictor* predicts the value of an example from the current parameter settings.  
*predictor\_string* gives a string representation of the predictor.

```

_____learnLinear.py — (continued) _____
40
41 def predictor(self,e):
42     """returns the prediction of the learner on example e"""
43     linpred = sum(w*f(e) for f,w in self.weights.items())
44     if self.squashed:
45         return sigmoid(linpred)
46     else:
47         return linpred
48
49 def predictor_string(self, sig_dig=3):
50     """returns the doc string for the current prediction function
51     sig_dig is the number of significant digits in the numbers"""
52     doc = "+".join(str(round(val,sig_dig))+ "*" + feat.__doc__
53                     for feat,val in self.weights.items())
54     if self.squashed:
55         return "sigmoid("+ doc +")"
56     else:
57         return doc

```

*learn* is the main algorithm of the learner. It does *num\_iter* steps of gradient descent. The other parameters it gets from the class.

```

_____learnLinear.py — (continued) _____
59 def learn(self,num_iter=100):
60     for it in range(num_iter):
61         self.display(2,"prediction=",self.predictor_string())
62         for e in self.train:
63             predicted = self.predictor(e)
64             error = self.target(e) - predicted
65             update = self.learning_rate*error
66             for feat in self.weights:
67                 self.weights[feat] += update*feat(e)
68             #self.predictor.__doc__ = self.predictor_string()
69             #return self.predictor

```

*one* is a function that always returns 1. This is used for one of the input properties.

```

_____learnLinear.py — (continued) _____
71 def one(e):
72     "1"
73     return 1

```

*sigmoid*(*x*) is the function

$$\frac{1}{1 + e^{-x}}$$

```

learnLinear.py — (continued)
75 def sigmoid(x):
76     return 1/(1+math.exp(-x))

```

The following tests the learner on a data sets. Uncomment the other data sets for different examples.

```

learnLinear.py — (continued)
78 from learnProblem import Data_set, Data_from_file
79 import matplotlib.pyplot as plt
80 def test(**args):
81     data = Data_from_file('data/SPECT.csv', target_index=0)
82     # data = Data_from_file('data/mail_reading.csv', target_index=-1)
83     # data = Data_from_file('data/carbool.csv', target_index=-1)
84     learner = Linear_learner(data,**args)
85     learner.learn()
86     print("function learned is", learner.predictor_string())
87     for ecrit in Data_set.evaluation_criteria:
88         test_error = data.evaluate_dataset(data.test, learner.predictor, ecrit)
89         print("    Average", ecrit, "error is", test_error)

```

The following plots the errors on the training and test sets as a function of the number of steps of gradient descent.

```

learnLinear.py — (continued)
91 def plot_steps(learner=None,
92               data = None,
93               criterion="sum_squares",
94               step=1,
95               num_steps=1000,
96               log_scale=True,
97               label=""):
98     """
99     plots the training and test error for a learner.
100     data is the
101     learner_class is the class of the learning algorithm
102     criterion gives the evaluation criterion plotted on the y-axis
103     step specifies how many steps are run for each point on the plot
104     num_steps is the number of points to plot
105     """
106     plt.ion()
107     plt.xlabel("step")
108     plt.ylabel("Average "+criterion+" error")
109     if log_scale:
110         plt.xscale('log') #plt.semilogx() #Makes a log scale
111     else:
112         plt.xscale('linear')
113     if data is None:
114         data = Data_from_file('data/holiday.csv', num_train=19, target_index=-1)
115         #data = Data_from_file('data/SPECT.csv', target_index=0)

```



```

117     # data = Data_from_file('data/mail_reading.csv', target_index=-1)
118     # data = Data_from_file('data/carbool.csv', target_index=-1)
119     random.seed(None) # reset seed
120     if learner is None:
121         learner = Linear_learner(data)
122     train_errors = []
123     test_errors = []
124     for i in range(1,num_steps+1,step):
125         test_errors.append(data.evaluate_dataset(data.test, learner.predictor, criterion))
126         train_errors.append(data.evaluate_dataset(data.train, learner.predictor, criterion))
127         learner.display(2, "Train error:",train_errors[-1],
128                        "Test error:",test_errors[-1])
129         learner.learn(num_iter=step)
130     plt.plot(range(1,num_steps+1,step),train_errors,ls='-',c='k',label="training errors")
131     plt.plot(range(1,num_steps+1,step),test_errors,ls='--',c='k',label="test errors")
132     plt.legend()
133     plt.draw()
134     learner.display(1, "Train error:",train_errors[-1],
135                   "Test error:",test_errors[-1])
136
137 if __name__ == "__main__":
138     test()
139
140 # This generates the figure
141 # from learnProblem import Data_set_augmented,prod_feat
142 # data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0)
143 # dataplus = Data_set_augmented(data,[],[prod_feat])
144 # plot_steps(data=data,num_steps=10000)
145 # plot_steps(data=dataplus,num_steps=10000) # warning very slow

```

**Exercise 7.8** The squashed learner only makes predictions in the range  $(0,1)$ . If the output values are  $\{1,2,3,4\}$  there is no use prediction 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'.

The following plots the prediction as a function of the function of the number of steps of gradient descent. We first define a version of *range* that allows for real numbers (integers and floats).

```

learnLinear.py — (continued)
146 def arange(start,stop,step):
147     """returns enumeration of values in the range [start,stop) separated by step.
148     like the built-in range(start,stop,step) but allows for integers and floats.
149     Note that rounding errors are expected with real numbers.
150     """
151     while start<stop:
152         yield start
153         start += step
154
155 def plot_prediction(learner=None,
156                   data = None,

```

```

157         minx = 0,
158         maxx = 5,
159         step_size = 0.01, # for plotting
160         label="function"):
161     plt.ion()
162     plt.xlabel("x")
163     plt.ylabel("y")
164     if data is None:
165         data = Data_from_file('data/simp_regr.csv', prob_test=0,
166                               boolean_features=False, target_index=-1)
167     if learner is None:
168         learner = Linear_learner(data,squashed=False)
169     learner.learning_rate=0.001
170     learner.learn(100)
171     learner.learning_rate=0.0001
172     learner.learn(1000)
173     learner.learning_rate=0.00001
174     learner.learn(10000)
175     learner.display(1,"function learned is", learner.predictor_string(),
176                    "error=",data.evaluate_dataset(data.train, learner.predictor, "sum_squares"))
177     plt.plot([e[0] for e in data.train],[e[-1] for e in data.train],"bo",label="data")
178     plt.plot(list(arange(minx,maxx,step_size)),[learner.predictor(x)]
179            for x in arange(minx,maxx,step_size)],
180            label=label)
181     plt.legend()
182     plt.draw()

```

learnLinear.py — (continued)

```

184 from learnProblem import Data_set_augmented, power_feat
185 def plot_polynomials(data=None,
186                     learner_class = Linear_learner,
187                     max_degree=5,
188                     minx = 0,
189                     maxx = 5,
190                     num_iter = 100000,
191                     learning_rate = 0.0001,
192                     step_size = 0.01, # for plotting
193                     ):
194     plt.ion()
195     plt.xlabel("x")
196     plt.ylabel("y")
197     if data is None:
198         data = Data_from_file('data/simp_regr.csv', prob_test=0,
199                               boolean_features=False, target_index=-1)
200     plt.plot([e[0] for e in data.train],[e[-1] for e in data.train],"ko",label="data")
201     x_values = list(arange(minx,maxx,step_size))
202     line_styles = ['-','--','-.-',':']
203     colors = ['0.5','k','k','k','k']
204     for degree in range(max_degree):
205         data_aug = Data_set_augmented(data,[power_feat(n) for n in range(1,degree+1)],

```

```

206         include_orig=False)
207     learner = learner_class(data_aug, squashed=False)
208     learner.learning_rate=learning_rate
209     learner.learn(num_iter)
210     learner.display(1, "For degree", degree,
211                    "function learned is", learner.predictor_string(),
212                    "error=", data.evaluate_dataset(data.train, learner.predictor, "sum_squares"))
213     ls = line_styles[degree % len(line_styles)]
214     col = colors[degree % len(colors)]
215     plt.plot(x_values, [learner.predictor([x]) for x in x_values], linestyle=ls, color=col,
216              label="degree="+str(degree))
217     plt.legend(loc='upper left')
218     plt.draw()
219
220 # Try:
221 # plot_prediction()
222 # plot_polynomials()
223 #data = Data_from_file('data/mail_reading.csv', target_index=-1)
224 #plot_prediction(data=data)

```

### 7.5.1 Batched Stochastic Gradient Descent

This implements batched stochastic gradient descent. If the batch size is 1, it can be simplified by not storing the differences in  $d$ , but applying them directly; this would be equivalent to the original code!

This overrides the learner *Linear\_learner*. Note that the comparison with regular gradient descent is unfair as the number of updates per step is not the same. (How could it be made more fair?)

```

_____learnLinearBSGD.py — Linear Learner with Batched Stochastic Gradient Descent_____
11 from learnLinear import Linear_learner
12 import random, math
13
14 class Linear_learner_bsgd(Linear_learner):
15     def __init__(self, *args, batch_size=10, **kargs):
16         Linear_learner.__init__(self, *args, **kargs)
17         self.batch_size = batch_size
18
19     def learn(self, num_iter=None):
20         if num_iter is None:
21             num_iter = self.number_iterations
22         batch_size = min(self.batch_size, len(self.train))
23         d = {feat:0 for feat in self.weights}
24         for it in range(num_iter):
25             self.display(2, "prediction=", self.predictor_string())
26             for e in random.sample(self.train, batch_size):
27                 predicted = self.predictor(e)
28                 error = self.target(e) - predicted
29                 update = self.learning_rate*error

```

```

30         for feat in self.weights:
31             d[feat] += update*feat(e)
32         for feat in self.weights:
33             self.weights[feat] += d[feat]
34             d[feat]=0
35
36 # from learnLinear import plot_steps
37 # from learnProblem import Data_from_file
38 # data = Data_from_file('data/holiday.csv', target_index=-1)
39 # learner = Linear_learner_bsgd(data)
40 # plot_steps(learner = learner, data=data)
41
42 # to plot polynomials with batching (compare to SGD)
43 # from learnLinear import plot_polynomials
44 # plot_polynomials(learner_class = Linear_learner_bsgd)

```

## 7.6 Deep Neural Network Learning

This provides a modular implementation that implements the layers modularly. Layers can easily be configured in many configurations. A layer needs to implement a function to compute the output values from the inputs and a way to back-propagate the error.

```

learnNN.py — Neural Network Learning
11 from learnProblem import Learner, Data_set, Data_from_file
12 from learnLinear import sigmoid, one
13 import random, math
14
15 class Layer(object):
16     def __init__(self,nn,num_outputs=None):
17         """Given a list of inputs, outputs will produce a list of length num_outputs.
18         nn is the neural network this is part of
19         num outputs is the number of outputs for this layer.
20         """
21         self.nn = nn
22         self.num_inputs = nn.num_outputs # output of nn is the input to this layer
23         if num_outputs:
24             self.num_outputs = num_outputs
25         else:
26             self.num_outputs = nn.num_outputs # same as the inputs
27
28     def output_values(self,input_values):
29         """Return the outputs for this layer for the given input values.
30         input_values is a list of the inputs to this layer (of length num_inputs)
31         returns a list of length self.num_outputs
32         """
33         raise NotImplementedError("output_values") # abstract method
34
35     def backprop(self,errors):

```

```

36         """Backpropagate the errors on the outputs, return the errors on the inputs.
37         errors is a list of errors for the outputs (of length self.num_outputs).
38         Return the errors for the inputs to this layer (of length self.num_inputs).
39         You can assume that this is only called after corresponding output_values,
40         and it can remember information information required for the backpropagation.
41         """
42         raise NotImplementedError("backprop") # abstract method

```

A linear layer maintains an array of weights. `self.weights[o][i]` is the weight between input  $i$  and output  $o$ . A 1 is added to the inputs.

learnNN.py — (continued)

```

44 class Linear_complete_layer(Layer):
45     """a completely connected layer"""
46     def __init__(self, nn, num_outputs, max_init=0.2):
47         """A completely connected linear layer.
48         nn is a neural network that the inputs come from
49         num_outputs is the number of outputs
50         max_init is the maximum value for random initialization of parameters
51         """
52         Layer.__init__(self, nn, num_outputs)
53         # self.weights[o][i] is the weight between input i and output o
54         self.weights = [[random.uniform(-max_init, max_init)
55                         for inf in range(self.num_inputs+1)]
56                        for outf in range(self.num_outputs)]
57
58     def output_values(self, input_values):
59         """Returns the outputs for the input values.
60         It remembers the values for the backprop.
61
62         Note in self.weights there is a weight list for every output,
63         so wts in self.weights effectively loops over the outputs.
64         """
65         self.inputs = input_values + [1]
66         return [sum(w*val for (w,val) in zip(wts,self.inputs))
67                for wts in self.weights]
68
69     def backprop(self, errors):
70         """Backpropagate the errors, updating the weights and returning the error in its inputs.
71         """
72         input_errors = [0]*(self.num_inputs+1)
73         for out in range(self.num_outputs):
74             for inp in range(self.num_inputs+1):
75                 input_errors[inp] += self.weights[out][inp] * errors[out]
76                 self.weights[out][inp] += self.nn.learning_rate * self.inputs[inp] * errors[out]
77         return input_errors[:-1] # remove the error for the "1"

```

learnNN.py — (continued)

```

79 class Sigmoid_layer(Layer):
80     """sigmoids of the inputs.
81     The number of outputs is equal to the number of inputs.

```

```

82     Each output is the sigmoid of its corresponding input.
83     """
84     def __init__(self, nn):
85         Layer.__init__(self, nn)
86
87     def output_values(self, input_values):
88         """Returns the outputs for the input values.
89         It remembers the output values for the backprop.
90         """
91         self.outputs= [sigmoid(inp) for inp in input_values]
92         return self.outputs
93
94     def backprop(self, errors):
95         """Returns the derivative of the errors"""
96         return [e*out*(1-out) for e,out in zip(errors, self.outputs)]

```

---

learnNN.py — (continued)

---

```

98 class ReLU_layer(Layer):
99     """Rectified linear unit (ReLU)  $f(z) = \max(0, z)$ .
100     The number of outputs is equal to the number of inputs.
101     """
102     def __init__(self, nn):
103         Layer.__init__(self, nn)
104
105     def output_values(self, input_values):
106         """Returns the outputs for the input values.
107         It remembers the input values for the backprop.
108         """
109         self.input_values = input_values
110         self.outputs= [max(0,inp) for inp in input_values]
111         return self.outputs
112
113     def backprop(self, errors):
114         """Returns the derivative of the errors"""
115         return [e if inp>0 else 0 for e,inp in zip(errors, self.input_values)]

```

---

learnNN.py — (continued)

---

```

117 class NN(Learner):
118     def __init__(self, dataset, learning_rate=0.1):
119         self.dataset = dataset
120         self.learning_rate = learning_rate
121         self.input_features = dataset.input_features
122         self.num_outputs = len(self.input_features)
123         self.layers = []
124
125     def add_layer(self, layer):
126         """add a layer to the network.
127         Each layer gets values from the previous layer.
128         """
129         self.layers.append(layer)

```

```

130         self.num_outputs = layer.num_outputs
131
132     def predictor(self,ex):
133         """Predicts the value of the first output feature for example ex.
134         """
135         values = [f(ex) for f in self.input_features]
136         for layer in self.layers:
137             values = layer.output_values(values)
138         return values[0]
139
140     def predictor_string(self):
141         return "not implemented"

```

The *test* method learns a network and evaluates it according to various criteria.

---

```

learnNN.py — (continued)
143
144     def learn(self,num_iter):
145         """Learns parameters for a neural network using stochastic gradient decent.
146         num_iter is the number of iterations
147         """
148         for i in range(num_iter):
149             for e in random.sample(self.dataset.train,len(self.dataset.train)):
150                 # compute all outputs
151                 values = [f(e) for f in self.input_features]
152                 for layer in self.layers:
153                     values = layer.output_values(values)
154                 # backpropagate
155                 errors = self.sum_squares_error([self.dataset.target(e)],values)
156                 for layer in reversed(self.layers):
157                     errors = layer.backprop(errors)
158
159     def sum_squares_error(self,observed,predicted):
160         """Returns the errors for each of the target features.
161         """
162         return [obsd-pred for obsd,pred in zip(observed,predicted)]

```

This constructs a neural network consisting of neural network with one hidden later. The hidden using used a ReLU activation function. The output layer used a sigmoid.

---

```

learnNN.py — (continued)
165 data = Data_from_file('data/mail_reading.csv', target_index=-1)
166 #data = Data_from_file('data/mail_reading_consis.csv', target_index=-1)
167 #data = Data_from_file('data/SPECT.csv', probab_test=0.5, target_index=0)
168 #data = Data_from_file('data/holiday.csv', target_index=-1) #, num_train=19)
169 nn1 = NN(data)
170 nn1.add_layer(Linear_complete_layer(nn1,3))
171 nn1.add_layer(Sigmoid_layer(nn1)) # comment this or the next
172 # nn1.add_layer(ReLU_layer(nn1))
173 nn1.add_layer(Linear_complete_layer(nn1,1))

```

```

174 | nn1.add_layer(Sigmoid_layer(nn1))
175 | nn1.learning_rate=0.1
176 | #nn1.learn(100)
177 |
178 | from learnLinear import plot_steps
179 | import time
180 | start_time = time.perf_counter()
181 | plot_steps(learner = nn1, data = data, num_steps=10000)
182 | for eg in data.train:
183 |     print(eg,nn1.predictor(eg))
184 | end_time = time.perf_counter()
185 | print("Time:", end_time - start_time)

```

**Exercise 7.9** In the definition of *nn1* above, for each of the following, first hypothesize what will happen, then test your hypothesis, then explain whether your testing confirms your hypothesis or not. Test it for more than one data set, and use more than one run for each data set.

- Which fits the data better, having a sigmoid layer or a ReLU layer after the first linear layer?
- Which is faster, having a sigmoid layer or a ReLU layer after the first linear layer?
- 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?
- What happens if you have neither the sigmoid layer nor a ReLU layer after the first linear layer?
- What happens if you have a ReLU layer then a sigmoid layer after the first linear layer and before the second linear layer?

**Exercise 7.10** Do some

It is even possible to define a perceptron layer. Warning: you may need to change the learning rate to make this work. Should I add it into the code? It doesn't follow the official line.

```

class PerceptronLayer(Layer):
    def __init__(self, nn):
        Layer.__init__(self, nn)

    def output_values(self, input_values):
        """Returns the outputs for the input values.
        """
        self.outputs= [1 if inp>0 else -1 for inp in input_values]
        return self.outputs

    def backprop(self, errors):
        """Pass the errors through"""
        return errors

```



## 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 save the new dataset, but generates it as needed. The amount of space used is constant, independent on the size of the data set.

```

learnBoosting.py — Functional Gradient Boosting
11 from learnProblem import Data_set, Learner
12
13 class Boosted_dataset(Data_set):
14     def __init__(self, base_dataset, offset_fun):
15         """new dataset which is like base_dataset,
16            but offset_fun(e) is subtracted from the target of each example e
17         """
18         self.base_dataset = base_dataset
19         self.offset_fun = offset_fun
20         Data_set.__init__(self, base_dataset.train, base_dataset.test,
21                           base_dataset.prob_test, base_dataset.target_index)
22
23     def create_features(self):
24         self.input_features = self.base_dataset.input_features
25         def newout(e):
26             return self.base_dataset.target(e) - self.offset_fun(e)
27         newout.frange = self.base_dataset.target.frange
28         self.target = newout

```

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)
30 class Boosting_learner(Learner):
31     def __init__(self, dataset, base_learner_class):
32         self.dataset = dataset
33         self.base_learner_class = base_learner_class
34         mean = sum(self.dataset.target(e)
35                   for e in self.dataset.train)/len(self.dataset.train)
36         self.predictor = lambda e:mean # function that returns mean for each example
37         self.predictor.__doc__ = "lambda e:"+str(mean)
38         self.offsets = [self.predictor]
39         self.errors = [data.evaluate_dataset(data.test, self.predictor, "sum_squares")]
40         self.display(1,"Predict mean test set error=", self.errors[0] )
41
42
43     def learn(self, num_ensemble=10):
44         """adds num_ensemble learners to the ensemble.
45         returns a new predictor.
46         """
47         for i in range(num_ensemble):
48             train_subset = Boosted_dataset(self.dataset, self.predictor)

```

```

49         learner = self.base_learner_class(train_subset)
50         new_offset = learner.learn()
51         self.offsets.append(new_offset)
52         def new_pred(e, old_pred=self.predictor, off=new_offset):
53             return old_pred(e)+off(e)
54         self.predictor = new_pred
55         self.errors.append(data.evaluate_dataset(data.test, self.predictor,"sum_squares"))
56         self.display(1,"After Iteration",len(self.offsets)-1,"test set error=", self.errors[-1])
57     return self.predictor

```

For testing, *sp\_DT\_learner* returns a function that constructs a decision tree learner where the minimum number of examples is a proportion of the number of training examples. The value of 0.9 tends to have one split, and a value of 0.5 tends to have two splits (but test it). Thus this can be used to construct small decision trees that can be used as weak learners.

```

learnBoosting.py — (continued)
59 # Testing
60
61 from learnDT import DT_learner
62 from learnProblem import Data_set, Data_from_file
63
64 def sp_DT_learner(min_prop=0.9):
65     def make_learner(dataset):
66         mne = len(dataset.train)*min_prop
67         return DT_learner(dataset,min_number_examples=mne)
68     return make_learner
69
70 data = Data_from_file('data/carbool.csv', target_index=-1)
71 #data = Data_from_file('data/SPECT.csv', target_index=0)
72 #data = Data_from_file('data/mail_reading.csv', target_index=-1)
73 #data = Data_from_file('data/holiday.csv', num_train=19, target_index=-1)
74 learner9 = Boosting_learner(data, sp_DT_learner(0.9))
75 #learner7 = Boosting_learner(data, sp_DT_learner(0.7))
76 #learner5 = Boosting_learner(data, sp_DT_learner(0.5))
77 predictor9 = learner9.learn(10)
78 for i in learner9.offsets: print(i.__doc__)
79 import matplotlib.pyplot as plt
80
81 def plot_boosting(data,steps=10, thresholds=[0.5,0.1,0.01,0.001], markers=['-','--','-.',':']):
82     learners = [Boosting_learner(data, sp_DT_learner(th)) for th in thresholds]
83     predictors = [learner.learn(steps) for learner in learners]
84     plt.ion()
85     plt.xscale('linear') # change between log and linear scale
86     plt.xlabel("number of trees")
87     plt.ylabel(" error")
88     for (learner,(threshold,marker)) in zip(learners,zip(thresholds,markers)):
89         plt.plot(range(len(learner.errors)), learner.errors, ls=marker,c='k',
90                 label=str(round(threshold*100))+ "% min example threshold")
91     plt.legend()
92     plt.draw()

```

```

93 |
94 | # plot_boosting(data)

```

## 7.8 K-means

The k-means learner 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*[*i*][*c*] is sum of the values for the *i*'th feature *i* for members of class *c*. The average value of the *i*th feature in class *i* is

$$\frac{\text{feature\_sum}[i][c]}{\text{class\_counts}[c]}$$

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     def __init__(self, dataset, num_classes):
17         self.dataset = dataset
18         self.num_classes = num_classes
19         self.random_initialize()
20
21     def random_initialize(self):
22         # class_counts[c] is the number of examples with class=c
23         self.class_counts = [0]*self.num_classes
24         # feature_sum[i][c] is the sum of the values of feature i for class c
25         self.feature_sum = [[0]*self.num_classes
26                               for feat in self.dataset.input_features]
27         for eg in self.dataset.train:
28             cl = random.randrange(self.num_classes) # assign eg to random class
29             self.class_counts[cl] += 1
30             for (ind, feat) in enumerate(self.dataset.input_features):
31                 self.feature_sum[ind][cl] += feat(eg)
32         self.num_iterations = 0
33         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)
35 def distance(self,cl,eg):
36     """distance of the eg from the mean of the class"""
37     return sum( (self.class_prediction(ind,cl)-feat(eg))*2
38                 for (ind,feat) in enumerate(self.dataset.input_features))
39
40 def class_prediction(self,feat_ind,cl):
41     """prediction of the class cl on the feature with index feat_ind"""
42     if self.class_counts[cl] == 0:
43         return 0 # there are no examples so we can choose any value
44     else:
45         return self.feature_sum[feat_ind][cl]/self.class_counts[cl]
46
47 def class_of_eg(self,eg):
48     """class to which eg is assigned"""
49     return (min((self.distance(cl,eg),cl)
50                 for cl in range(self.num_classes)))[1]
51     # 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)
53 def k_means_step(self):
54     """Updates the model with one step of k-means.
55     Returns whether the assignment is stable.
56     """
57     new_class_counts = [0]*self.num_classes
58     # feature_sum[i][c] is the sum of the values of feature i for class c
59     new_feature_sum = [[0]*self.num_classes
60                         for feat in self.dataset.input_features]
61     for eg in self.dataset.train:
62         cl = self.class_of_eg(eg)
63         new_class_counts[cl] += 1
64         for (ind,feat) in enumerate(self.dataset.input_features):
65             new_feature_sum[ind][cl] += feat(eg)
66     stable = (new_class_counts == self.class_counts) and (self.feature_sum == new_feature_sum)
67     self.class_counts = new_class_counts
68     self.feature_sum = new_feature_sum
69     self.num_iterations += 1
70     return stable
71
72
73 def learn(self,n=100):
74     """do n steps of k-means, or until convergence"""
75     i=0
76     stable = False
77     while i<n and not stable:

```

```

78         stable = self.k_means_step()
79         i += 1
80         self.display(1,"Iteration",self.num_iterations,
81                     "class counts: ",self.class_counts," Stable=",stable)
82     return stable
83
84     def show_classes(self):
85         """sorts the data by the class and prints in order.
86         For visualizing small data sets
87         """
88         class_examples = [[] for i in range(self.num_classes)]
89         for eg in self.dataset.train:
90             class_examples[self.class_of_eg(eg)].append(eg)
91         print("Class", "Example", sep='\t')
92         for cl in range(self.num_classes):
93             for eg in class_examples[cl]:
94                 print(cl,*eg,sep='\t')
95
96     def plot_error(self, maxstep=20):
97         """Plots the sum-of-squares error as a function of the number of steps"""
98         plt.ion()
99         plt.xlabel("step")
100        plt.ylabel("Ave sum-of-squares error")
101        train_errors = []
102        if self.dataset.test:
103            test_errors = []
104        for i in range(maxstep):
105            self.learn(1)
106            train_errors.append( sum(self.distance(self.class_of_eg(eg),eg)
107                                   for eg in self.dataset.train)
108                               /len(self.dataset.train))
109            if self.dataset.test:
110                test_errors.append( sum(self.distance(self.class_of_eg(eg),eg)
111                                       for eg in self.dataset.test)
112                                   /len(self.dataset.test))
113        plt.plot(range(1,maxstep+1),train_errors,
114                label=str(self.num_classes)+" classes. Training set")
115        if self.dataset.test:
116            plt.plot(range(1,maxstep+1),test_errors,
117                    label=str(self.num_classes)+" classes. Test set")
118        plt.legend()
119        plt.draw()
120
121    data = Data_from_file('data/emdata2.csv', num_train=10, target_index=2000)
122    kml = K_means_learner(data,2)
123    num_iter=4
124    print("Class assignment after",num_iter,"iterations:")
125    kml.learn(num_iter); kml.show_classes()
126
127    # Plot the error

```

```

128 # km2=K_means_learner(data,2); km2.plot_error(20) # 2 classes
129 # km3=K_means_learner(data,3); km3.plot_error(20) # 3 classes
130 # km13=K_means_learner(data,13); km13.plot_error(20) # 13 classes
131
132 # data = Data_from_file('data/carbool.csv', target_index=2000,boolean_features=True)
133 # km1 = K_means_learner(data,3)
134 # km1.learn(20); km1.show_classes()
135 # km3=K_means_learner(data,3); km3.plot_error(20) # 3 classes
136 # km3=K_means_learner(data,30); km3.plot_error(20) # 30 classes

```

**Exercise 7.11** Change *boolean\_features = True* flag to allow for numerical features. K-means assumes the features are numerical, so we want to make non-numerical features into numerical features (using characteristic functions) but we probably don't want to change numerical features into Boolean.

**Exercise 7.12** 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:

- Initialize the classes with actual examples, so that the classes will not start empty. (Do the classes become empty?)
- 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.

## 7.9 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  $i$ th 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
11 from learnProblem import Data_set, Learner, Data_from_file
12 import random
13 import math
14 import matplotlib.pyplot as plt
15
16 class EM_learner(Learner):
17     def __init__(self, dataset, num_classes):
18         self.dataset = dataset
19         self.num_classes = num_classes
20         self.class_counts = None
21         self.feature_counts = None

```

The function *em\_step* goes through 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)
23 def em_step(self, orig_class_counts, orig_feature_counts):
24     """updates the model."""
25     class_counts = [0]*self.num_classes
26     feature_counts = [{val:[0]*self.num_classes
27                        for val in feat.frange}
28                      for feat in self.dataset.input_features]
29     for tple in self.dataset.train:
30         if orig_class_counts: # a model exists
31             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 for a tuple, given the current statistics.

$$\begin{aligned}
 P(c \mid tple) &\propto P(c) * \prod_i P(X_i = tple(i) \mid c) \\
 &= \frac{class\_counts[c]}{len(self.dataset)} * \prod_i \frac{feature\_counts[i][feat_i(tple)][c]}{class\_counts[c]}
 \end{aligned}$$

$len(self.dataset)$  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 the original tuple in the current model
42     """
43     feats = self.dataset.input_features

```

```

44     unnorm = [prod(feature_counts[i][feat(tple)][c]
45                  for (i,feat) in enumerate(feats))/(class_counts[c]**(len(feats)-1))
46                  for c in range(self.num_classes)]
47     thesum = sum(unnorm)
48     return [un/thesum for un in unnorm]

```

*learn* does  $n$  steps of EM:

```

learnEM.py — (continued)
50     def learn(self,n):
51         """do n steps of em"""
52         for i in range(n):
53             self.class_counts,self.feature_counts = self.em_step(self.class_counts,
54                                                                    self.feature_counts)

```

The following is for visualizing the classes. It prints the dataset ordered by the probability of class  $c$ .

```

learnEM.py — (continued)
56     def show_class(self,c):
57         """sorts the data by the class and prints in order.
58         For visualizing small data sets
59         """
60         sorted_data = sorted((self.prob(tpl,self.class_counts,self.feature_counts)[c],
61                               ind, # preserve ordering for equal probabilities
62                               tpl)
63                               for (ind,tpl) in enumerate(self.dataset.train))
64         for cc,r,tpl in sorted_data:
65             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:

$$\begin{aligned}
 P(tple) &= \sum_c P(c) * \prod_i P(X_i=tple(i) | c) \\
 &= \sum_c \frac{cc[c]}{\text{len}(self.dataset)} * \prod_i \frac{fc[i][feat_i(tple)][c]}{cc[c]}
 \end{aligned}$$

where  $cc$  is the class count and  $fc$  is feature count.  $\text{len}(self.dataset)$  can be distributed out of the sum, and  $cc[c]$  can be taken out of the product:

$$= \frac{1}{\text{len}(self.dataset)} \sum_c \frac{1}{cc[c]^{\#feats-1}} * \prod_i fc[i][feat_i(tple)][c]$$

Given the probability of each tuple, we can evaluate the logloss, as the negative of the log probability:

```

learnEM.py — (continued)
67     def logloss(self,tple):
68         """returns the logloss of the prediction on tple, which is -log(P(tple))
69         based on the current class counts and feature counts

```



```

70     """
71     feats = self.dataset.input_features
72     res = 0
73     cc = self.class_counts
74     fc = self.feature_counts
75     for c in range(self.num_classes):
76         res += prod(fc[i][feat(tple)][c]
77                     for (i, feat) in enumerate(feats))/(cc[c]**(len(feats)-1))
78     if res>0:
79         return -math.log2(res/len(self.dataset.train))
80     else:
81         return float("inf") #infinity
82
83 def plot_error(self, maxstep=20):
84     """Plots the logloss error as a function of the number of steps"""
85     plt.ion()
86     plt.xlabel("step")
87     plt.ylabel("Ave Logloss (bits)")
88     train_errors = []
89     if self.dataset.test:
90         test_errors = []
91     for i in range(maxstep):
92         self.learn(1)
93         train_errors.append( sum(self.logloss(tple) for tple in self.dataset.train)
94                             /len(self.dataset.train))
95         if self.dataset.test:
96             test_errors.append( sum(self.logloss(tple) for tple in self.dataset.test)
97                                /len(self.dataset.test))
98     plt.plot(range(1,maxstep+1),train_errors,
99             label=str(self.num_classes)+" classes. Training set")
100    if self.dataset.test:
101        plt.plot(range(1,maxstep+1),test_errors,
102                label=str(self.num_classes)+" classes. Test set")
103    plt.legend()
104    plt.draw()
105
106 def prod(L):
107     """returns the product of the elements of L"""
108     res = 1
109     for e in L:
110         res *= e
111     return res
112
113 def random_dist(k):
114     """generate k random numbers that sum to 1"""
115     res = [random.random() for i in range(k)]
116     s = sum(res)
117     return [v/s for v in res]
118
119 data = Data_from_file('data/emdata2.csv', num_train=10, target_index=2000)

```

```

120 | eml = EM_learner(data,2)
121 | num_iter=2
122 | print("Class assignment after",num_iter,"iterations:")
123 | eml.learn(num_iter); eml.show_class(0)
124 |
125 | # Plot the error
126 | # em2=EM_learner(data,2); em2.plot_error(40) # 2 classes
127 | # em3=EM_learner(data,3); em3.plot_error(40) # 3 classes
128 | # em13=EM_learner(data,13); em13.plot_error(40) # 13 classes
129 |
130 | # data = Data_from_file('data/carbool.csv', target_index=2000,boolean_features=False)
131 | # [f.frange for f in data.input_features]
132 | # eml = EM_learner(data,3)
133 | # eml.learn(20); eml.show_class(0)
134 | # em3=EM_learner(data,3); em3.plot_error(60) # 3 classes
135 | # em3=EM_learner(data,30); em3.plot_error(60) # 30 classes

```

**Exercise 7.13** For the EM data, where there are naturally 2 classes, 3 classes does better on the training set after a while than 2 classes, but worse on the test set. Explain why. Hint: look what the 3 classes are. Use "em3.show\_class(i)" for each of the classes  $i \in [0,3)$ .

**Exercise 7.14** Write code to plot the logloss as a function of the number of classes (from 1 to say 15) for a fixed number of iterations. (From the experience with the existing code, think about how many iterations is appropriate.)

## Reasoning Under Uncertainty

### 8.1 Representing Probabilistic Models

In the implementation of probabilistic models we will assume that the variables are objects, rather than the strings we used for CSPs. (Note that in the CSP code variables could be anything; we just used strings for the examples.) We use a class here because it is more amenable to extend to richer models, such as when we introduce time.

A variable consists of a name and a domain. The domain of a variable is a list or a tuple, as the ordering will matter in the representation of factors. The code below internally uses the index of each value. We define a function *val\_to\_index* that maps from the value to the index.

```
_____probVariables.py — Probabilistic Variables_____
11 class Variable(object):
12     """A random variable.
13     name (string) - name of the variable
14     domain (list) - a list of the values for the variable.
15     Variables are ordered according to their name.
16     """
17
18     def __init__(self, name, domain):
19         self.name = name
20         self.size = len(domain)
21         self.domain = domain
22         self.val_to_index = {} # map from domain to index
23         for i, val in enumerate(domain):
24             self.val_to_index[val]=i
25
26     def __str__(self):
27         return self.name
```

<i>A</i>	<i>B</i>	<i>C</i>	Value
0	a	s	$v_0$
0	a	t	$v_1$
0	b	s	$v_2$
0	b	t	$v_3$
0	c	s	$v_4$
0	c	t	$v_5$
1	a	s	$v_6$
1	a	t	$v_7$
1	b	s	$v_8$
1	b	t	$v_9$
1	c	s	$v_{10}$
1	c	t	$v_{11}$

Figure 8.1: A representation for a factor for the variable ordering  $A, B, C$ 

```

28
29     def __repr__(self):
30         return "Variable('"+self.name+"')"
```

## 8.2 Factors

Factors are functions from variables into values. The main problem with variable elimination is the amount of space used, because it saves the intermediate factors. (If instead it recomputed factors rather than saving the factors, it would be effectively enumerating the worlds, and so would be exponential in the number of variables). We only want to store the list of numbers, with as little bookkeeping as possible.

A total ordering of the variables, and a total ordering of the values in the domains of the variables induces a total ordering of the values of the factor according to the lexicographic ordering. E.g., suppose the domain of  $A$  is  $[0, 1]$ , domain of  $B$  is  $[a', b', c']$ , and the domain of  $C$  is  $[s', t']$ , the ordering  $[A, B, C]$  of variables induces an ordering on the values of the factor, as in Figure 8.1.

We just need to store the list of variables and the  $v_i$ s. For any assignment to  $A$ ,  $B$  and  $C$ , we can compute the index of the value for that assignment.  $A = a, B = b, C = c$  is stored at location  $a' * 6 + b' * 2 + c'$ , where  $a'$  is  $A.val\_to\_index[a]$ , and similarly for  $b'$  and  $c'$ .

```

_____probFactors.py — Factor manipulation for graphical models_____
11 from functools import reduce
12 #from probVariables import Variable
13
14 class Factor(object):
15     nextid=0 # each factor has a unique identifier; for printing
```

```

16
17 def __init__(self, variables):
18     """variables is the ordered list of variables
19     """
20     self.variables = variables # ordered list of variables
21     # Compute the size and the offsets for the variables
22     self.var_offsets = {}
23     self.size = 1
24     for i in range(len(variables)-1, -1, -1):
25         self.var_offsets[variables[i]] = self.size
26         self.size *= variables[i].size
27     self.id = Factor.nextid
28     Factor.nextid += 1

```

For each factor, *get\_value* returns the value of the factor for an assignment. An **assignment** is a variable:value dictionary. The assignment must include all of the variables involved in the factor, and can include variables not in the factor. This needs to be defined for every subclass.

```

_____probFactors.py — (continued)_____
30 def get_value(self, assignment):
31     raise NotImplementedError("get_value") # abstract method

```

The methods *str* and *brief* return string representations of the factor, as a table or just as a name with the variables it is a factor on.

```

_____probFactors.py — (continued)_____
33 def __str__(self, variables=None):
34     """returns a string representation of the factor.
35     Allows for an arbitrary variable ordering.
36     variables is a list of the variables in the factor
37     (can contain other variables)"""
38     if variables==None:
39         variables = self.variables
40     else:
41         variables = [v for v in variables if v in self.variables]
42     res = ""
43     for v in variables:
44         res += str(v) + "\t"
45     res += "f"+str(self.id)+"\n"
46     for i in range(self.size):
47         asst = self.index_to_assignment(i)
48         for v in variables:
49             res += str(asst[v])+"\t"
50         res += str(self.get_value(asst))
51         res += "\n"
52     return res
53
54 def brief(self):
55     """returns a string representing a summary of the factor"""
56     res = "f"+str(self.id)+"("

```

```

57     for i in range(0, len(self.variables)-1):
58         res += str(self.variables[i])+", "
59     if len(self.variables)>0:
60         res += str(self.variables[len(self.variables)-1])
61     res += ")"
62     return res

```

The methods *assignment\_to\_index* and *index\_to\_assignment* map between the assignments of values to variables and the index of where that assignment would be stored.

```

_____probFactors.py — (continued) _____
64     def assignment_to_index(self, assignment):
65         """returns the index where the variable:value assignment is stored"""
66         index = 0
67         for var in self.variables:
68             index += var.val_to_index[assignment[var]]*self.var_offsets[var]
69         return index
70
71     def index_to_assignment(self, index):
72         """gives a dict representation of the variable assignment for index
73         """
74         asst = {}
75         for i in range(len(self.variables)-1, -1, -1):
76             asst[self.variables[i]] = self.variables[i].domain[index % self.variables[i].size]
77             index = index // self.variables[i].size
78         return asst

```

A *Factor\_stored* is a factor that has the values stored in a list.

```

_____probFactors.py — (continued) _____
80     class Factor_stored(Factor):
81         def __init__(self, variables, values):
82             Factor.__init__(self, variables)
83             self.values = values
84
85         def get_value(self, assignment):
86             return self.values[self.assignment_to_index(assignment)]

```

A *Factor\_observed* 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) _____
88     class Factor_observed(Factor):
89         def __init__(self, factor, obs):
90             Factor.__init__(self, [v for v in factor.variables if v not in obs])
91             self.observed = obs
92             self.orig_factor = factor
93

```

```

94 |     def get_value(self, assignment):
95 |         ass = assignment.copy()
96 |         for ob in self.observed:
97 |             ass[ob]=self.observed[ob]
98 |         return self.orig_factor.get_value(ass)

```

A *Factor\_sum* is a factor that is the result of summing out a variable from the product of other factors. Ie., it constructs a representation of:

$$\sum_{var} \prod_{f \in factors} f.$$

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) \_

```

100 | class Factor_sum(Factor_stored):
101 |     def __init__(self, var, factors):
102 |         self.var_summed_out = var
103 |         self.factors = factors
104 |         self.vars = []
105 |         for fac in factors:
106 |             for v in fac.variables:
107 |                 if v is not var and v not in self.vars:
108 |                     self.vars.append(v)
109 |         Factor_stored.__init__(self, self.vars, None)
110 |         self.values = [None]*self.size
111 |
112 |     def get_value(self, assignment):
113 |         """lazy implementation: if not saved, compute it. Return saved value"""
114 |         index = self.assignment_to_index(assignment)
115 |         if self.values[index]:
116 |             return self.values[index]
117 |         else:
118 |             total = 0
119 |             new_asst = assignment.copy()
120 |             for val in self.var_summed_out.domain:
121 |                 new_asst[self.var_summed_out] = val
122 |                 prod = 1
123 |                 for fac in self.factors:
124 |                     prod *= fac.get_value(new_asst)
125 |                 total += prod
126 |             self.values[index] = total
127 |         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*.

```

129 def factor_times(variable,factors):
130     """when factors are factors just on variable (or on no variables)"""
131     prods= []
132     facts = [f for f in factors if variable in f.variables]
133     for val in variable.domain:
134         prod = 1
135         ast = {variable:val}
136         for f in facts:
137             prod *= f.get_value(ast)
138         prods.append(prod)
139     return prods

```

*Prob* is a factor that represents a conditional probability.

```

_____probFactors.py — (continued)_____
141 class Prob(Factor_stored):
142     """A factor defined by a conditional probability table"""
143     def __init__(self,var,pars,cpt):
144         """Creates a factor from a conditional probability table, cptf.
145         The cpt values are assumed to be for the ordering par+[var]
146         """
147         Factor_stored.__init__(self,pars+[var],cpt)
148         self.child = var
149         self.parents = pars
150         assert self.size==len(cpt),"Table size incorrect "+str(self)

```

*cond\_dist* returns the probability distribution of the child given values from the parent. This code is based on *assignment\_to\_index*. Similarly, *cont\_prob* returns the probability that the child has a particular value given an assignment of values to the parents. In both of these *par\_assignment* is a dict that assigns all of the parents (and can also assign other variables, but these are ignored).

```

_____probFactors.py — (continued)_____
152 def cond_dist(self,par_assignment):
153     """returns the distribution (a val:prob dictionary) over the child given
154     assignment to the parents
155
156     par_assignment is a variable:value dictionary that assigns values to parents
157     """
158     index = 0
159     for var in self.parents:
160         index += var.val_to_index[par_assignment[var]]*self.var_offsets[var]
161     # index is the position where the distribution starts
162     return {self.child.domain[i]:self.values[index+i] for i in range(len(self.child.domain))}
163
164 def cond_prob(self,par_assignment,child_value):
165     """returns the probability child has child_value given
166     assignment to the parents
167
168     par_assignment is a variable:value dictionary that assigns values to parents
169     child_value is a value to the child

```



```

170         """
171         index = self.child.val_to_index[child_value]
172         for var in self.parents:
173             index += var.val_to_index[par_assignment[var]]*self.var_offsets[var]
174         return self.values[index]

```

A *Factor\_rename* is a factor that is the result 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 the all variables are renamed.

```

_____probFactors.py — (continued)_____
176 class Factor_rename(Factor):
177     def __init__(self, fac, renaming):
178         Factor.__init__(self, list(renaming.keys()))
179         self.orig_fac = fac
180         self.renaming = renaming
181
182     def get_value(self, assignment):
183         return self.orig_fac.get_value({self.renaming[var]:val
184                                         for (var, val) in assignment.items()
185                                         if var in self.variables})

```

## 8.3 Graphical Models

A graphical model consists of a set of variables and a set of factors. A belief network is a graphical model where all of the factors represent conditional probabilities. There are some operations (such as pruning variables) which are applicable to belief networks, but are not applicable to more general models. At the moment, we will treat them as the same.

```

_____probGraphicalModels.py — Graphical Models and Belief Networks_____
11 class Graphical_model(object):
12     """The class of graphical models.
13     A graphical model consists of a set of variables and a set of factors.
14
15     List vars is a list of variables
16     List factors is a list of factors
17     """
18     def __init__(self, vars=None, factors=None):
19         self.variables = vars
20         self.factors = factors

```

A belief network is a graphical model where all of the factors are conditional probabilities, and every variable has a conditional probability. This only checks the first condition:

```

_____probGraphicalModels.py — (continued)_____
22 class Belief_network(Graphical_model):

```

```

23     """The class of belief networks."""
24
25     def __init__(self, vars=None, factors=None):
26         """vars is a list of variables
27         factors is a list of factors. Here we assume that all of the factors are instances of Prob.
28         """
29         Graphical_model.__init__(self, vars, factors)
30         assert all(isinstance(f, Prob) for f in factors) if factors else True

```

Each of the inference methods implements the query method that computes the posterior probability of a variable given a dictionary of variable:value observations. These are all Displayable because they implement the *display* method which is currently text-based.

```

_____probGraphicalModels.py — (continued) _____
32 from utilities import Displayable
33
34 class Inference_method(Displayable):
35     """The abstract class of graphical model inference methods"""
36     def query(self, qvar, obs={}):
37         raise NotImplementedError("Inference_method query") # abstract method

```

The first example belief network is a simple chain  $A \rightarrow B \rightarrow C$ .

```

_____probGraphicalModels.py — (continued) _____
39 from probVariables import Variable
40 from probFactors import Prob
41
42 boolean = [False, True]
43 A = Variable("A", boolean)
44 B = Variable("B", boolean)
45 C = Variable("C", boolean)
46
47 f_a = Prob(A, [], [0.4, 0.6])
48 f_b = Prob(B, [A], [0.9, 0.1, 0.2, 0.8])
49 f_c = Prob(C, [B], [0.5, 0.5, 0.3, 0.7])
50
51 bn1 = Belief_network([A, B, C], [f_a, f_b, f_c])

```

The second Bayesian network is the report-of-leaving example from Poole and Mackworth, Artificial Intelligence, 2010 <http://artint.info>. This is Example 6.10 (page 236) shown in Figure 6.1.

```

_____probGraphicalModels.py — (continued) _____
53 # Bayesian network report of leaving example from
54 # Poole and Mackworth, Artificial Intelligence, 2010 http://artint.info
55 # This is Example 6.10 (page 236) shown in Figure 6.1
56
57 Al = Variable("Alarm", boolean)
58 Fi = Variable("Fire", boolean)
59 Le = Variable("Leaving", boolean)

```

```

60 Re = Variable("Report", boolean)
61 Sm = Variable("Smoke", boolean)
62 Ta = Variable("Tamper", boolean)
63
64 f_ta = Prob(Ta,[],[0.98,0.02])
65 f-fi = Prob(Fi,[],[0.99,0.01])
66 f-sm = Prob(Sm,[Fi],[0.99,0.01,0.1,0.9])
67 f-al = Prob(Al,[Fi,Ta],[0.9999, 0.0001, 0.15, 0.85, 0.01, 0.99, 0.5, 0.5])
68 f-lv = Prob(Le,[Al],[0.999, 0.001, 0.12, 0.88])
69 f-re = Prob(Re,[Le],[0.99, 0.01, 0.25, 0.75])
70
71 bn2 = Belief_network([Al,Fi,Le,Re,Sm,Ta],[f_ta,f-fi,f-sm,f-al,f-lv,f-re])

```

The third Bayesian network is the sprinkler example from Pearl.

```

_____probGraphicalModels.py — (continued)_____
73
74 Season = Variable("Season",["summer","winter"])
75 Sprinkler = Variable("Sprinkler",["on","off"])
76 Rained = Variable("Rained",boolean)
77 Grass_wet = Variable("Grass wet",boolean)
78 Grass_shiny = Variable("Grass shiny",boolean)
79 Shoes_wet = Variable("Shoes wet",boolean)
80
81 f_season = Prob(Season,[],[0.5,0.5])
82 f_sprinkler = Prob(Sprinkler,[Season],[0.9,0.1,0.05,0.95])
83 f_rained = Prob(Rained,[Season],[0.7,0.3,0.2,0.8])
84 f_wet = Prob(Grass_wet,[Sprinkler,Rained], [1,0,0.1,0.9,0.2,0.8,0.02,0.98])
85 f_shiny = Prob(Grass_shiny, [Grass_wet], [0.95,0.05,0.3,0.7])
86 f_shoes = Prob(Shoes_wet, [Grass_wet], [0.92,0.08,0.35,0.65])
87
88 bn3 = Belief_network([Season, Sprinkler, Rained, Grass_wet, Grass_shiny, Shoes_wet],
89                      [f_season, f_sprinkler, f_rained, f_wet, f_shiny, f_shoes])

```

## 8.4 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, Factor_observed, Factor_sum, factor_times
12 from probGraphicalModels import Graphical_model, Inference_method
13
14 class VE(Inference_method):
15     """The class that queries Graphical Models using variable elimination.
16
17     gm is graphical model to query
18     """

```

```

19     def __init__(self, gm=None):
20         self.gm = gm
21
22     def query(self, var, obs={}, elim_order=None):
23         """computes P(var|obs) where
24         var is a variable
25         obs is a variable:value dictionary"""
26         if var in obs:
27             return [1 if val == obs[var] else 0 for val in var.domain]
28         else:
29             if elim_order == None:
30                 elim_order = self.gm.variables
31             projFactors = [self.project_observations(fact, obs)
32                           for fact in self.gm.factors]
33             for v in elim_order:
34                 if v != var and v not in obs:
35                     projFactors = self.eliminate_var(projFactors, v)
36             unnorm = factor_times(var, projFactors)
37             p_obs = sum(unnorm)
38             self.display(1, "Unnormalized probs:", unnorm, "Prob obs:", p_obs)
39             return {val: pr/p_obs for val, pr in zip(var.domain, unnorm)}

```

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 of assigning a value to a single variable.

---

probVE.py — (continued)

---

```

41     def project_observations(self, factor, obs):
42         """Returns the resulting factor after observing obs
43
44         obs is a dictionary of variable:value pairs.
45         """
46         if any((var in obs) for var in factor.variables):
47             # a variable in factor is observed
48             return Factor_observed(factor, obs)
49         else:
50             return factor
51
52     def eliminate_var(self, factors, var):
53         """Eliminate a variable var from a list of factors.
54         Returns a new set of factors that has var summed out.
55         """
56         self.display(2, "eliminating ", str(var))
57         contains_var = []
58         not_contains_var = []
59         for fac in factors:
60             if var in fac.variables:
61                 contains_var.append(fac)
62             else:
63                 not_contains_var.append(fac)

```

```

64         if contains_var == []:
65             return factors
66         else:
67             newFactor = Factor_sum(var,contains_var)
68             self.display(2,"Multiplying:",[f.brief() for f in contains_var])
69             self.display(2,"Creating factor:", newFactor.brief())
70             self.display(3,"Factor in detail", newFactor)
71             not_contains_var.append(newFactor)
72             return not_contains_var
73
74 from probGraphicalModels import bn1, A,B,C
75 bn1v = VE(bn1)
76 ## bn1v.query(A,{})
77 ## bn1v.query(C,{})
78 ## Inference_method.max_display_level = 3 # show more detail in displaying
79 ## Inference_method.max_display_level = 1 # show less detail in displaying
80 ## bn1v.query(A,{C:True})
81 ## bn1v.query(B,{A:True,C:False})
82
83 from probGraphicalModels import bn2,Al,Fi,Le,Re,Sm,Ta
84 bn2v = VE(bn2) # answers queries using variable elimination
85 ## bn2v.query(Ta,{})
86 ## Inference_method.max_display_level = 0 # show no detail in displaying
87 ## bn2v.query(Le,{})
88 ## bn2v.query(Ta,{},elim_order=[Sm,Re,Le,Al,Fi])
89 ## bn2v.query(Ta,{Re:True})
90 ## bn2v.query(Ta,{Re:True,Sm:False})
91
92 from probGraphicalModels import bn3, Season, Sprinkler, Rained, Grass_wet, Grass_shiny, Shoes_wet
93 bn3v = VE(bn3)
94 ## bn3v.query(Shoes_wet,{})
95 ## bn3v.query(Shoes_wet,{Rained:True})
96 ## bn3v.query(Shoes_wet,{Grass_shiny:True})
97 ## bn3v.query(Shoes_wet,{Grass_shiny:False,Rained:True})

```

## 8.5 Stochastic Simulation

### 8.5.1 Sampling from a discrete distribution

The method *sample\_one* generates a single sample from a (possible 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_____
11 import random
12 from probGraphicalModels import Inference_method
13
14 def sample_one(dist):
15     """returns the index of a single sample from normalized distribution dist."""

```

```

16     rand = random.random()*sum(dist.values())
17     cum = 0    # cumulative weights
18     for v in dist:
19         cum += dist[v]
20         if cum > rand:
21             return v

```

If we want to generate multiple samples, repeatedly calling *sample\_one* may not be efficient. If we want to generate  $n$  samples, and the distribution is over  $m$  values, *sample\_one* takes time  $O(mn)$ . If  $m$  and  $n$  are of the same order of magnitude, we can do better.

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 cannot all be 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)
23 def sample_multiple(dist, num_samples):
24     """returns a list of num_samples values selected using distribution dist.
25     dist is a value:weight dictionary that does not need to be normalized
26     """
27     total = sum(dist.values())
28     rands = sorted(random.random()*total for i in range(num_samples))
29     result = []
30     dist_items = list(dist.items())
31     cum = dist_items[0][1] # cumulative sum
32     index = 0
33     for r in rands:
34         while r>cum:
35             index += 1
36             cum += dist_items[index][1]
37         result.append(dist_items[index][0])
38     return result

```

### Exercise 8.1

What is the time and space complexity the following 4 methods to generate  $n$  samples, where  $m$  is the length of *dist*:

- $n$  calls to *sample\_one*
- sample\_multiple*
- 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.
- Choose a random number in the range  $[i/n, (i+1)/n)$  for each  $i \in \text{range}(n)$ , 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 few samples and also for many samples.

```

_____probStochSim.py — (continued) _____
40 def test_sampling(dist, num_samples):
41     """Given a distribution, dist, draw num_samples samples
42     and return the resulting counts
43     """
44     result = {v:0 for v in dist}
45     for v in sample_multiple(dist, num_samples):
46         result[v] += 1
47     return result
48
49 # try the following queries a number of times each:
50 # test_sampling({1:1,2:2,3:3,4:4}, 100)
51 # test_sampling({1:1,2:2,3:3,4:4}, 100000)

```

### 8.5.2 Sampling Methods for Belief Network Inference

A *Sampling\_inference\_method* is an *Inference\_method*, 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 Bayesian network (and not an undirected graphical model).

```

_____probStochSim.py — (continued) _____
53 class Sampling_inference_method(Inference_method):
54     """The abstract class of sampling-based belief network inference methods"""
55     def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
56         raise NotImplementedError("Sampling_inference_method query") # abstract

```

Some of the sampling methods require a sample order of factors representing conditional probabilities, where the parents of a node must come before the node in the sample order. The following method computes such a sample ordering, and is used when the *sample\_order* argument is *None*.

```

_____probStochSim.py — (continued) _____
58 def select_sample_ordering(bn):
59     """creates a sample ordering of factors such that the parents of a node
60     are before the node.
61     raises StopIteration if there is no such ordering. This would occur in next().
62     """
63     sample_order=[]
64     defined = set() # set of variables whose probability is defined
65     factors_to_sample = bn.factors.copy()
66     while factors_to_sample:
67         fac = next(f for f in factors_to_sample
68                 if all(par in defined for par in f.parents))

```

```

69         factors_to_sample.remove(fac)
70         sample_order.append(fac)
71         defined.add(fac.child)
72     return sample_order

```

### 8.5.3 Rejection Sampling

```

_____probStochSim.py — (continued)_____
74 class Rejection_sampling(Sampling_inference_method):
75     """The class that queries Graphical Models using Rejection Sampling.
76
77     bn is a belief network to query
78     """
79     def __init__(self, bn=None):
80         self.bn = bn
81         self.label = "Rejection Sampling"
82
83     def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
84         """computes P(qvar|obs) where
85         qvar is a variable.
86         obs is a variable:value dictionary.
87         sample_order is a list of factors where factors defining the parents
88         come before the factors for the child.
89         """
90         if sample_order is None:
91             sample_order = select_sample_ordering(self.bn)
92         self.display(2, *[f.child for f in sample_order], sep="\t")
93         counts = {val:0 for val in qvar.domain}
94         for i in range(number_samples):
95             rejected = False
96             sample = {}
97             for fac in sample_order:
98                 nvar = fac.child #next variable
99                 val = sample_one(fac.cond_dist(sample))
100                 self.display(2, val, end="\t")
101                 if nvar in obs and obs[nvar] != val:
102                     rejected = True
103                     self.display(2, "Rejected")
104                     break
105                 sample[nvar] = val
106             if not rejected:
107                 counts[sample[qvar]] += 1
108                 self.display(2, "Accepted")
109         tot = sum(counts.values())
110         return counts, {c:divide(v,tot) for (c,v) in counts.items()}

```

It is possible that all samples get rejected. In that case, Python would give as a arithmetic error. Instead, we implement the convention that  $0/0 = 1$ . You need to be careful is using these numbers as probabilities.



```

112 def divide(num,denom):
113     """returns num/denom without divide-by-zero errors.
114     defines 0/0 to be 1."""
115     if denom == 0:
116         return 1.0
117     else:
118         return num/denom

```

### 8.5.4 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.

```

120 class Likelihood_weighting(Sampling_inference_method):
121     """The class that queries Graphical Models using Likelihood weighting.
122
123     bn is a belief network to query
124     """
125     def __init__(self,bn=None):
126         self.bn = bn
127         self.label = "Likelihood weighting"
128
129     def query(self,qvar,obs={},number_samples=1000,sample_order=None):
130         """computes P(qvar|obs) where
131         qvar is a variable.
132         obs is a variable:value dictionary.
133         sample_order is a list of factors where factors defining the parents
134         come before the factors for the child.
135         """
136         if sample_order is None:
137             sample_order = select_sample_ordering(self.bn)
138         self.display(2,*[f.child for f in sample_order
139                        if f.child not in obs],sep="\t")
140         counts = [0 for val in qvar.domain]
141         for i in range(number_samples):
142             sample = {}
143             weight = 1.0
144             for fac in sample_order:
145                 nvar = fac.child # next variable sampled
146                 if nvar in obs:
147                     sample[nvar] = obs[nvar]
148                     weight *= fac.get_value(sample)
149                 else:
150                     val = sample_one(fac.cond_dist(sample))
151                     self.display(2,val,end="\t")
152                     sample[nvar] = val
153             counts[sample[qvar]] += weight

```

```

154         self.display(2,weight)
155         tot = sum(counts)
156         return counts, {c:v/tot for (c,v) in counts.items()}

```

**Exercise 8.2** Change this algorithm so that it does **importance sampling** using a proposal distribution. It needs *sample\_one* using a different distribution and then update the weight of the current sample. For testing, use a proposal distribution that only specifies probabilities for some of the variables (and the algorithm uses the probabilities for the network in other cases).

### 8.5.5 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 need to be copied during resampling.

```

probStochSim.py — (continued)
158 class Particle_filtering(Sampling_inference_method):
159     """The class that queries Graphical Models using Particle Filtering.
160
161     bn is a belief network to query
162     """
163     def __init__(self,bn=None):
164         self.bn = bn
165         self.label = "Particle Filtering"
166
167     def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
168         """computes P(qvar|obs) where
169         qvar is a variable.
170         obs is a variable:value dictionary.
171         sample_order is a list of factors where factors defining the parents
172         come before the factors for the child.
173         """
174         if sample_order is None:
175             sample_order = select_sample_ordering(self.bn)
176         self.display(2,*[f.child for f in sample_order
177                        if f.child not in obs],sep="\t")
178         particles = [{ } for i in range(number_samples)]
179         for fac in sample_order:
180             nvar = fac.child # the variable sampled
181             if nvar in obs:
182                 weights = {part:fac.cond_prob(part,obs[nvar]) for part in particles}
183                 particles = [p.copy for p in resample(particles, weights, number_samples)]
184             else:
185                 for part in particles:
186                     part[nvar] = sample_one(fac.cond_dist(part))
187                 self.display(2,part[nvar],end="\t")
188         counts = [0 for val in qvar.domain]
189         for part in particles:
190             counts[part[qvar]] += 1

```

```

191         self.display(2,weight)
192         return counts

```

### 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) _____
194 def resample(particles, weights, num_samples):
195     """returns num_samples copies of particles resampled according to weights.
196     particles is a list of particles
197     weights is a list of positive numbers, of same length as particles
198     num_samples is n integer
199     """
200     total = sum(weights)
201     rands = sorted(random.random()*total for i in range(num_samples))
202     result = []
203     cum = weights[0]    # cumulative sum
204     index = 0
205     for r in rands:
206         while r>cum:
207             index += 1
208             cum += weights[index]
209         result.append(particles[index])
210     return result

```

### 8.5.6 Examples

```

_____probStochSim.py — (continued) _____
212 from probGraphicalModels import bn1, A,B,C
213 bn1r = Rejection_sampling(bn1)
214 bn1L = Likelihood_weighting(bn1)
215 ## Inference_method.max_display_level = 2 # detailed tracing for all inference methods
216 ## bn1r.query(A,{})
217 ## bn1r.query(C,{})
218 ## bn1r.query(A,{C:True})
219 ## bn1r.query(B,{A:True,C:False})
220
221 from probGraphicalModels import bn2,A1,Fi,Le,Re,Sm,Ta
222 bn2r = Rejection_sampling(bn2) # answers queries using rejection sampling
223 bn2L = Likelihood_weighting(bn2) # answers queries using rejection sampling
224 bn2p = Particle_filtering(bn2) # answers queries using particle filtering
225 ## bn2r.query(Ta,{})
226 ## bn2r.query(Ta,{})
227 ## bn2r.query(Ta,{Re:True})
228 ## Inference_method.max_display_level = 0 # no detailed tracing for all inference methods

```

```

229 ## bn2r.query(Ta,{Re:True},number_samples=100000)
230 ## bn2r.query(Ta,{Re:True,Sm:False})
231 ## bn2r.query(Ta,{Re:True,Sm:False},number_samples=100)
232
233 ## bn2L.query(Ta,{Re:True,Sm:False},number_samples=100)
234 ## bn2L.query(Ta,{Re:True,Sm:False},number_samples=100)
235
236
237 from probGraphicalModels import bn3,Season, Sprinkler
238 from probGraphicalModels import Rained, Grass_wet, Grass_shiny, Shoes_wet
239 bn3r = Rejection_sampling(bn3) # answers queries using rejection sampling
240 bn3L = Likelihood_weighting(bn3) # answers queries using rejection sampling
241 bn3p = Particle_filtering(bn3) # answers queries using particle filtering
242 #bn3r.query(Shoes_wet,{Grass_shiny:True,Rained:True})
243 #bn3L.query(Shoes_wet,{Grass_shiny:True,Rained:True})
244 #bn3p.query(Shoes_wet,{Grass_shiny:True,Rained:True})

```

**Exercise 8.3** This code keeps regenerating the distribution of a variable given its parents. Implement one or both of the following, and compare them to the original. Make *cond\_dist* return a slice that corresponds to the distribution, and then use the slice instead of the dictionary (a list slice does not generate new data structures). Make *cond\_dist* remember values it has already computed, and only return these.

### 8.5.7 Plotting Behaviour 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 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 *what* is “*prob\_ev*”, the probability of evidence.

probStochSim.py — (continued)

```

246 import matplotlib.pyplot as plt

```

```

247
248 def plot_stats(method, what, qvar, obs, number_samples=100, number_runs=1000):
249     """Plots a cumulative distribution of the prediction of the model.
250     method is a Sampling_inference_method (that implements appropriate query())
251     what is either "prob_ev" or the value of qvar to plot
252     qvar is the query variable
253     obs is the variable:value dictionary representing the observations
254     number_samples is the number of samples for each run
255     number_iterations is the number of runs that are plotted
256     """
257     plt.ion()
258     plt.xlabel("value")
259     plt.ylabel("Cumulative Number")
260     Inference_method.max_display_level, prev_max_display_level = 0, Inference_method.max_display_level
261     answers = [method.query(qvar, obs, number_samples=number_samples)
262                for i in range(number_runs)]
263     if what == "prob_ev":
264         values = [sum(ans)/number_samples for ans in answers]
265         label = method.label+"(prob of evidence)"
266     else:
267         values = [divide(ans[qvar.val_to_index[what]],sum(ans)) for ans in answers]
268         label = method.label+" (" +str(qvar)+"="+str(what)+")"
269     values.sort()
270     plt.plot(values, range(number_runs), label=label)
271     plt.legend(loc="upper left")
272     plt.draw()
273     Inference_method.max_display_level = prev_max_display_level # restore display level
274
275
276 # plot_stats(bn2r, False, Ta, {Re: True, Sm: False}, number_samples=1000, number_runs=1000)
277 # plot_stats(bn2L, False, Ta, {Re: True, Sm: False}, number_samples=1000, number_runs=1000)
278 # plot_stats(bn2r, False, Ta, {Re: True, Sm: False}, number_samples=100, number_runs=1000)
279 # plot_stats(bn2L, False, Ta, {Re: True, Sm: False}, number_samples=100, number_runs=1000)
280 # plot_stats(bn3r, True, Shoes_wet, {Grass_shiny: True, Rained: True}, number_samples=1000)
281 # plot_stats(bn3L, True, Shoes_wet, {Grass_shiny: True, Rained: True}, number_samples=1000)
282 # plot_stats(bn2r, "prob_ev", Ta, {Re: True, Sm: False}, number_samples=1000, number_runs=1000)
283 # plot_stats(bn2L, "prob_ev", Ta, {Re: True, Sm: False}, number_samples=1000, number_runs=1000)

```

## 8.6 Markov Chain Monte Carlo

The following implements **Gibbs sampling**, a form of **Markov Chain Monte Carlo** MCMC.

```

_____probMCMC.py — Markov Chain Monte Carlo (Gibbs sampling)_____
11 import random
12 from probGraphicalModels import Inference_method
13
14 from probStochSim import sample_one, Sampling_inference_method
15

```

```

16 class Gibbs_sampling(Sampling_inference_method):
17     """The class that queries Graphical Models using Gibbs Sampling.
18
19     bn is a graphical model (e.g., a belief network) to query
20     """
21     def __init__(self, bn=None):
22         self.bn = bn
23         self.label = "Gibbs Sampling"
24
25     def query(self, qvar, obs={}, number_samples=1000, burn_in=100, sample_order=None):
26         """computes P(qvar|obs) where
27         qvar is a variable.
28         obs is a variable:value dictionary.
29         sample_order is a list of non-observed variables in order.
30         """
31         counts = {val:0 for val in qvar.domain}
32         if sample_order is not None:
33             variables = sample_order
34         else:
35             variables = [v for v in self.bn.variables if v not in obs]
36         var_to_factors = {v:set() for v in self.bn.variables}
37         for fac in self.bn.factors:
38             for var in fac.variables:
39                 var_to_factors[var].add(fac)
40         sample = {var:random.choice(var.domain) for var in variables}
41         self.display(2, "Sample:", sample)
42         sample.update(obs)
43         for i in range(burn_in + number_samples):
44             if sample_order == None:
45                 random.shuffle(variables)
46             for var in variables:
47                 # get probability distribution of var given its neighbours
48                 vardist = {val:1 for val in var.domain}
49                 for val in var.domain:
50                     sample[var] = val
51                     for fac in var_to_factors[var]: # Markov blanket
52                         vardist[val] *= fac.get_value(sample)
53                 sample[var] = sample_one(vardist)
54             if i >= burn_in:
55                 counts[sample[qvar]] +=1
56         tot = sum(counts.values())
57         return counts, {c:v/tot for (c,v) in counts.items()}
58
59 from probGraphicalModels import bn1, A,B,C
60 bn1g = Gibbs_sampling(bn1)
61 ## Inference_method.max_display_level = 2 # detailed tracing for all inference methods
62 bn1g.query(A,{})
63 ## bn1g.query(C,{})
64 ## bn1g.query(A,{C:True})
65 ## bn1g.query(B,{A:True,C:False})

```

```

66
67 from probGraphicalModels import bn2, Al, Fi, Le, Re, Sm, Ta
68 bn2g = Gibbs_sampling(bn2)
69 ## bn2g.query(Ta, {Re: True}, number_samples=100000)

```

**Exercise 8.4** 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 8.5** 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?

## 8.7 Hidden Markov Models

This code for hidden Markov models is independent of the graphical models code, to keep it simple. Section 8.8 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_____
11 import random
12 from probStochSim import sample_one, sample_multiple
13
14 class HMM(object):
15     def __init__(self, states, obsvars, pobs, trans, indist):
16         """A hidden Markov model.
17         states - set of states
18         obsvars - set of observation variables
19         pobs - probability of observations, pobs[i][s] is P(Obs_i=True | State=s)
20         trans - transition probability - trans[i][j] gives P(State=j | State=i)
21         indist - initial distribution - indist[s] is P(State_0 = s)
22         """
23         self.states = states
24         self.obsvars = obsvars
25         self.pobs = pobs
26         self.trans = trans
27         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)
29 # state
30 #     0=middle, 1,2,3 are corners
31 states1 = {'middle', 'c1', 'c2', 'c3'} # states
32 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)
34 # pobs gives the observation model:
35 #pobs[mi][state] is P(mi=on | state)
36 closeMic=0.6; farMic=0.1; midMic=0.4
37 pobs1 = {'m1':{'middle':midMic, 'c1':closeMic, 'c2':farMic, 'c3':farMic}, # mic 1
38         'm2':{'middle':midMic, 'c1':farMic, 'c2':closeMic, 'c3':farMic}, # mic 2
39         '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)
41 # trans specifies the dynamics
42 # trans[i] is the distribution over states resulting from state i
43 # trans[i][j] gives P(S=j | S=i)
44 sm=0.7; mmc=0.1 # transition probabilities when in middle
45 sc=0.8; mcm=0.1; mcc=0.05 # transition probabilities when in a corner
46 trans1 = {'middle':{'middle':sm, 'c1':mmc, 'c2':mmc, 'c3':mmc}, # was in middle
47          'c1':{'middle':mcm, 'c1':sc, 'c2':mcc, 'c3':mcc}, # was in corner 1
48          'c2':{'middle':mcm, 'c1':mcc, 'c2':sc, 'c3':mcc}, # was in corner 2
49          '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)
51 # initially we have a uniform distribution over the animal's state
52 indist1 = {st:1.0/len(states1) for st in states1}
53
54 hmm1 = HMM(states1, obs1, pobs1, trans1, indist1)

```

### 8.7.1 Exact Filtering for HMMs

A *HMM.VE\_filter* has a current state distribution which can be updated by observing or by advancing to the next time.



```

56 from utilities import Displayable
57
58 class HMM_VE_filter(Displayable):
59     def __init__(self, hmm):
60         self.hmm = hmm
61         self.state_dist = hmm.indist
62
63     def filter(self, obsseq):
64         """updates and returns the state distribution following the sequence of
65         observations in obsseq using variable elimination.
66
67         Note that it first advances time.
68         This is what is required if it is called sequentially.
69         If that is not what is wanted initially, do an observe first.
70         """
71         for obs in obsseq:
72             self.advance() # advance time
73             self.observe(obs) # observe
74         return self.state_dist
75
76     def observe(self, obs):
77         """updates state conditioned on observations.
78         obs is a list of values for each observation variable"""
79         for i in self.hmm.obsvars:
80             self.state_dist = {st:self.state_dist[st]*(self.hmm.pobs[i][st]
81                                                         if obs[i] else (1-self.hmm.pobs[i][st]))
82                               for st in self.hmm.states}
83         norm = sum(self.state_dist.values()) # normalizing constant
84         self.state_dist = {st:self.state_dist[st]/norm for st in self.hmm.states}
85         self.display(2, "After observing", obs, "state distribution:", self.state_dist)
86
87     def advance(self):
88         """advance to the next time"""
89         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

```

The following are some queries for *hmm1*.

```

95 hmm1f1 = HMM_VE_filter(hmm1)
96 # hmm1f1.filter([{'m1':0, 'm2':1, 'm3':1}, {'m1':1, 'm2':0, 'm3':1}])
97 ## HMM_VE_filter.max_display_level = 2 # show more detail in displaying
98 # hmm1f2 = HMM_VE_filter(hmm1)
99 # hmm1f2.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':1, 'm3':0}, {'m1':1, 'm2':0, 'm3':0},
100 #               {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
101 #               {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':1}, {'m1':0, 'm2':0, 'm3':1},
102 #               {'m1':0, 'm2':0, 'm3':1}])

```

```

103 # hmm1f3 = HMM_VE_filter(hmm1)
104 # hmm1f3.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':1, 'm2':0, 'm3':0}, {'m1':
105
106 # How do the following differ in the resulting state distribution?
107 # Note they start the same, but have different initial observations.
108 ## HMM_VE_filter.max_display_level = 1 # show less detail in displaying
109 # for i in range(100): hmm1f1.advance()
110 # hmm1f1.state_dist
111 # for i in range(100): hmm1f3.advance()
112 # hmm1f3.state_dist

```

**Exercise 8.6** 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. Change the code to allow for controlled HMMs. Hint: the action only influences the state transition.

**Exercise 8.7** 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.

### 8.7.2 Particle Filtering for HMMs

In this implementation a particle is just a state. If you want to do some form of smooting, 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) _____
113 from utilities import Displayable
114 from probStochSim import resample
115
116 class HMM_particle_filter(Displayable):
117     def __init__(self, hmm, number_particles=1000):
118         self.hmm = hmm
119         self.particles = [sample_one(hmm.indist)
120                           for i in range(number_particles)]
121         self.weights = [1 for i in range(number_particles)]
122
123     def filter(self, obsseq):
124         """returns the state distribution following the sequence of
125         observations in obsseq using particle filtering.
126
127         Note that it first advances time.
128         This is what is required if it is called after previous filtering.
129         If that is not what is wanted initially, do an observe first.
130         """
131         for obs in obsseq:
132             self.advance() # advance time
133             self.observe(obs) # observe

```

```

134         self.resample_particles()
135         self.display(2, "After observing", str(obs),
136                     "state distribution:", self.histogram(self.particles))
137     self.display(1, "Final state distribution:", self.histogram(self.particles))
138     return self.histogram(self.particles)
139
140     def advance(self):
141         """advance to the next time.
142         This assumes that all of the weights are 1."""
143         self.particles = [sample_one(self.hmm.trans[st])
144                           for st in self.particles]
145
146     def observe(self, obs):
147         """reweight the particles to incorporate observations obs"""
148         for i in range(len(self.particles)):
149             for obv in obs:
150                 if obs[obv]:
151                     self.weights[i] *= self.hmm.pobs[obv][self.particles[i]]
152                 else:
153                     self.weights[i] *= 1-self.hmm.pobs[obv][self.particles[i]]
154
155     def histogram(self, particles):
156         """returns list of the probability of each state as represented by
157         the particles"""
158         tot=0
159         hist = {st: 0.0 for st in self.hmm.states}
160         for (st,wt) in zip(self.particles,self.weights):
161             hist[st]+=wt
162         tot += wt
163         return {st:hist[st]/tot for st in hist}
164
165     def resample_particles(self):
166         """resamples to give a new set of particles."""
167         self.particles = resample(self.particles, self.weights, len(self.particles))
168         self.weights = [1] * len(self.particles)

```

The following are some queries for *hmm1*.

---

```

170 |hmm1pf1 = HMM_particle_filter(hmm1)
171 |# HMM_particle_filter.max_display_level = 2 # show each step
172 |# hmm1pf1.filter([{'m1':0, 'm2':1, 'm3':1}, {'m1':1, 'm2':0, 'm3':1}])
173 |# hmm1pf2 = HMM_particle_filter(hmm1)
174 |# hmm1pf2.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':1, 'm3':0}, {'m1':1, 'm2':0, 'm3':0},
175 |#                 {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
176 |#                 {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':1}, {'m1':0, 'm2':0, 'm3':1},
177 |#                 {'m1':0, 'm2':0, 'm3':1}])
178 |# hmm1pf3 = HMM_particle_filter(hmm1)
179 |# hmm1pf3.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':1, 'm2':0, 'm3':0}, {'

```

**Exercise 8.8** 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 8.9** 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.

### 8.7.3 Generating Examples

The following code is useful for generating examples.

```

181  def simulate(hmm,horizon):
182      """returns a pair of (state sequence, observation sequence) of length horizon.
183      for each time t, the agent is in state_sequence[t] and
184      observes observation_sequence[t]
185      """
186      state = sample_one(hmm.indist)
187      obsseq=[]
188      stateseq=[]
189      for time in range(horizon):
190          stateseq.append(state)
191          newobs = {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
192                  for obs in hmm.obsvars}
193          obsseq.append(newobs)
194          state = sample_one(hmm.trans[state])
195      return stateseq,obsseq
196
197  def simobs(hmm,stateseq):
198      """returns observation sequence for the state sequence"""
199      obsseq=[]
200      for state in stateseq:
201          newobs = {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
202                  for obs in hmm.obsvars}
203          obsseq.append(newobs)
204      return obsseq
205
206  def create_eg(hmm,n):
207      """Create an annotated example for horizon n"""
208      seq,obs = simulate(hmm,n)
209      print("True state sequence:",seq)
210      print("Sequence of observations:\n",obs)
211      hmmfilter = HMM_VE_filter(hmm)
212      dist = hmmfilter.filter(obs)
213      print("Resulting distribution over states:\n",dist)

```

## 8.8 Dynamic Belief Networks

A dynamic belief network consists of:

- A set of features. A variable is a feature-time pair.
- An initial distribution over the features at time 0. This is a belief network with all variables being time 0 variables.
- A specification of the dynamics. Here we define the how the variables one time depend on variables at that time and the previous time, in such a way that the graph is acyclic.

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. However, the unrolled Bayesian network may be very large. We also need to construct multiple copies of each feature.
- Just representing the variables “now”. In this approach we can observe and query the current variables. We can then move to the next time. This does not allow for arbitrary historical queries (about the past or the future), but can be much simpler.

Here we will implement the second of these.

```

_____probDBN.py — Dynamic belief networks_____
11 from probVariables import Variable
12 from probGraphicalModels import Graphical_model
13 from probFactors import Prob, Factor_rename
14 from probVE import VE
15 from utilities import Displayable
16
17 class DBN_variable(Variable):
18     """A random variable that incorporates
19
20     A variable can have both a name and an index. The index defaults to 1.
21     Equality is true if they are both the name and the index are the same."""
22     def __init__(self, name, domain=[False, True], index=1):
23         Variable.__init__(self, name, domain)
24         self.index = index
25         self.previous = None
26
27     def __lt__(self, other):
28         if self.name != other.name:
29             return self.name < other.name

```

```

30         else:
31             return self.index < other.index
32
33     def __gt__(self, other):
34         return other < self
35
36     def __str__(self):
37         # if self.index == 1:
38         #     return self.name
39         # else:
40         return self.name + "_" + str(self.index)
41
42     __repr__ = __str__
43
44 def variable_pair(name, domain=[False, True]):
45     """returns a variable and its predecessor. This is used to define 2-stage DBNs
46
47     If the name is X, it returns the pair of variables X0, X"""
48     var = DBN_variable(name, domain)
49     var0 = DBN_variable(name, domain, index=0)
50     var.previous = var0
51     return var0, var

```

probDBN.py — (continued)

```

53 class DBN(Displayable):
54     """The class of stationary Dynamic Bayesian networks.
55
56     * vars1 is a list of current variables (each must have
57     previous variable).
58     * transition_factors is a list of factors for P(X|parents) where X
59     is a current variable and parents is a list of current or previous variables.
60     * init_factors is a list of factors for P(X|parents) where X is a
61     current variable and parents can only include current variables
62     The graph of transition factors + init factors must be acyclic.
63
64     """
65     def __init__(self, vars1, transition_factors=None, init_factors=None):
66         self.vars1 = vars1
67         self.vars0 = [v.previous for v in vars1]
68         self.transition_factors = transition_factors
69         self.init_factors = init_factors
70         self.var_index = {} # var_index[v] is the index of variable v
71         for i, v in enumerate(vars1):
72             self.var_index[v] = i

```

Here is a 3 variable DBN:

probDBN.py — (continued)

```

74 A0, A1 = variable_pair("A")
75 B0, B1 = variable_pair("B")
76 C0, C1 = variable_pair("C")

```

```

77
78 # dynamics
79 pc = Prob(C1,[B1,C0],[0.03,0.97,0.38,0.62,0.23,0.77,0.78,0.22])
80 pb = Prob(B1,[A0,A1],[0.5,0.5,0.77,0.23,0.4,0.6,0.83,0.17])
81 pa = Prob(A1,[A0,B0],[0.1,0.9,0.65,0.35,0.3,0.7,0.8,0.2])
82
83 # initial distribution
84 pa0 = Prob(A1,[],[0.9,0.1])
85 pb0 = Prob(B1,[A1],[0.3,0.7,0.8,0.2])
86 pc0 = Prob(C1,[],[0.2,0.8])
87
88 dbn1 = DBN([A1,B1,C1],[pa,pb,pc],[pa0,pb0,pc0])

```

Here is the animal example

```

_____probDBN.py — (continued)_____
90 from probHMM import closeMic, farMic, midMic, sm, mmc, sc, mcm, mcc
91
92 Pos_0,Pos_1 = variable_pair("Position",domain=[0,1,2,3])
93 Mic1_0,Mic1_1 = variable_pair("Mic1")
94 Mic2_0,Mic2_1 = variable_pair("Mic2")
95 Mic3_0,Mic3_1 = variable_pair("Mic3")
96
97 # conditional probabilities - see hmm for the values of sm,mmc, etc
98 ppos = Prob(Pos_1, [Pos_0],
99             [sm, mmc, mmc, mmc, #was in middle
100             mcm, sc, mcc, mcc, #was in corner 1
101             mcm, mcc, sc, mcc, #was in corner 2
102             mcm, mcc, mcc, sc]) #was in corner 3
103 pm1 = Prob(Mic1_1, [Pos_1], [1-midMic, midMic, 1-closeMic, closeMic,
104                             1-farMic, farMic, 1-farMic, farMic])
105 pm2 = Prob(Mic2_1, [Pos_1], [1-midMic, midMic, 1-farMic, farMic,
106                             1-closeMic, closeMic, 1-farMic, farMic])
107 pm3 = Prob(Mic3_1, [Pos_1], [1-midMic, midMic, 1-farMic, farMic,
108                             1-farMic, farMic, 1-closeMic, closeMic])
109 ipos = Prob(Pos_1,[], [0.25, 0.25, 0.25, 0.25])
110 dbn_an =DBN([Pos_1,Mic1_1,Mic2_1,Mic3_1],
111             [ppos, pm1, pm2, pm3],
112             [ipos, pm1, pm2, pm3])

```

```

_____probDBN.py — (continued)_____
114 class DBN_VE_filter(VE):
115     def __init__(self,dbn):
116         self.dbn = dbn
117         self.current_factors = dbn.init_factors
118         self.current_obs = {}
119
120     def observe(self, obs):
121         """updates the current observations with obs.
122         obs is a variable:value dictionary where variable is a current
123         variable.

```

```

124     """
125     assert all(self.current_obs[var]==obs[var] for var in obs
126               if var in self.current_obs),"inconsistent current observations"
127     self.current_obs.update(obs)
128
129     def query(self,var):
130         """returns the posterior probability of current variable var"""
131         return VE(Graphical_model(self.dbn.vars1,self.current_factors)).query(var,self.current_obs)
132
133     def advance(self):
134         """advance to the next time"""
135         prev_factors = [self.make_previous(fac) for fac in self.current_factors]
136         prev_obs = {var.previous:val for var,val in self.current_obs.items()}
137         two_stage_factors = prev_factors + self.dbn.transition_factors
138         self.current_factors = self.elim_vars(two_stage_factors,self.dbn.vars0,prev_obs)
139         self.current_obs = {}
140
141     def make_previous(self,fac):
142         """Creates new factor from fac where the current variables in fac
143         are renamed to previous variables.
144         """
145         return Factor_rename(fac, {var.previous:var for var in fac.variables})
146
147     def elim_vars(self,factors, vars, obs):
148         for var in vars:
149             if var in obs:
150                 factors = [self.project_observations(fac,obs) for fac in factors]
151             else:
152                 factors = self.eliminate_var(factors, var)
153         return factors

```

Example queries:

```

155 df = DBN_VE_filter(dbn1)
156 #df.observe({B1:True}); df.advance(); df.observe({C1:False})
157 #df.query(B1)
158 #df.advance()
159 #df.query(B1)
160 dfa = DBN_VE_filter(dbn_an)
161 # dfa.observe({Mic1_1:0, Mic2_1:1, Mic3_1:1})
162 # dfa.advance()
163 # dfa.observe({Mic1_1:1, Mic2_1:0, Mic3_1:1})
164 # dfa.query(Pos_1)

```



## Planning Under Uncertainty

### 9.1 Decision Networks

The decision network code builds on the representation for belief networks of Chapter 8.

We first allow for factors that define the utility. Here the utility is a function of the variables in *vars*, and the table is a list that enumerates the values as in Section 8.2.

```
-----decnNetworks.py — Representations for Decision Networks-----
11 from probGraphicalModels import Graphical_model
12 from probFactors import Factor_stored
13 from probVariables import Variable
14 from probFactors import Prob
15
16 class Utility(Factor_stored):
17     """A factor defined by a utility"""
18     def __init__(self, vars, table):
19         """Creates a factor on vars from the table.
20         The table is ordered according to vars.
21         """
22         Factor_stored.__init__(self, vars, table)
23         assert self.size==len(table), "Table size incorrect "+str(self)
```

A decision variable is a 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.

```
-----decnNetworks.py — (continued)-----
25 class DecisionVariable(Variable):
26     def __init__(self, name, domain, parents):
```

```

27     Variable.__init__(self,name,domain)
28     self.parents = parents
29     self.all_vars = set(parents) | {self}

```

A decision network is a graphical model where the variables can be random variables or decision variables. In the factors we assume there is one utility factor.

```

----- decnNetworks.py --- (continued) -----
31 class DecisionNetwork(Graphical_model):
32     def __init__(self,vars=None,factors=None):
33         """vars is a list of variables
34         factors is a list of factors (instances of Prob and Utility)
35         """
36         Graphical_model.__init__(self,vars,factors)

```

VE.DN is the decision network analogue of variable elimination. You can optimize the decisions using *optimize*. Note that *optimize* requires a legal elimination ordering otherwise it will give an exception.

```

----- decnNetworks.py --- (continued) -----
38 from probFactors import factor_times, Factor_stored
39 from probVE import VE
40
41 class VE_DN(VE):
42     """Variable Elimination for Decision Networks"""
43     def __init__(self,dn=None):
44         """dn is a decision network"""
45         VE.__init__(self,dn)
46         self.dn = dn
47
48     def optimize(self,elim_order=None,obs={}):
49         if elim_order == None:
50             elim_order = self.gm.variables
51         policy = []
52         proj_factors = [self.project_observations(fact,obs)
53                         for fact in self.dn.factors]
54         for v in elim_order:
55             if isinstance(v,DecisionVariable):
56                 to_max = [fac for fac in proj_factors
57                           if v in fac.variables and set(fac.variables) <= v.all_vars]
58                 assert len(to_max)==1, "illegal variable order "+str(elim_order)+" at "+str(v)
59                 newFac = Factor_max(v, to_max[0])
60                 policy.append(newFac.decision_fun)
61                 proj_factors = [fac for fac in proj_factors if fac is not to_max[0]]+[newFac]
62                 self.display(2,"maximizing",v,"resulting factor",newFac.brief() )
63                 self.display(3,newFac)
64             else:
65                 proj_factors = self.eliminate_var(proj_factors, v)
66         assert len(proj_factors)==1,"Should there be only one element of proj_factors?"
67         value = proj_factors[0].get_value({})

```

```
68 |         return value,policy
```

```
-----decnNetworks.py --- (continued)-----
```

```
70 | class Factor_max(Factor_stored):
71 |     """A factor obtained by maximizing a variable in a factor.
72 |     Also builds a decision_function. This is based on Factor_sum.
73 |     """
74 |
75 |     def __init__(self, dvar, factor):
76 |         """dvar is a decision variable.
77 |         factor is a factor that contains dvar and only parents of dvar
78 |         """
79 |         self.dvar = dvar
80 |         self.factor = factor
81 |         vars = [v for v in factor.variables if v is not dvar]
82 |         Factor_stored.__init__(self,vars,None)
83 |         self.values = [None]*self.size
84 |         self.decision_fun = Factor_stored(vars,[None]*self.size)
85 |
86 |     def get_value(self,assignment):
87 |         """lazy implementation: if saved, return save value else compute it"""
88 |         index = self.assignment_to_index(assignment)
89 |         if self.values[index]:
90 |             return self.values[index]
91 |         else:
92 |             max_val = float("-inf") # -infinity
93 |             new_asst = assignment.copy()
94 |             for elt in self.dvar.domain:
95 |                 new_asst[self.dvar] = elt
96 |                 fac_val = self.factor.get_value(new_asst)
97 |                 if fac_val>max_val:
98 |                     max_val = fac_val
99 |                     best_elt = elt
100 |             self.values[index] = max_val
101 |             self.decision_fun.values[index] = best_elt
102 |         return max_val
```

The fire decision network of Figure 9.1 is represented as:

```
-----decnNetworks.py --- (continued)-----
```

```
104 | boolean = [False, True]
105 | Al = Variable("Alarm", boolean)
106 | Fi = Variable("Fire", boolean)
107 | Le = Variable("Leaving", boolean)
108 | Re = Variable("Report", boolean)
109 | Sm = Variable("Smoke", boolean)
110 | Ta = Variable("Tamper", boolean)
111 | SS = Variable("See Sm", boolean)
112 | CS = DecisionVariable("Ch Sm", boolean,{Re})
113 | Call = DecisionVariable("Call", boolean,{SS,CS,Re})
114 |
```

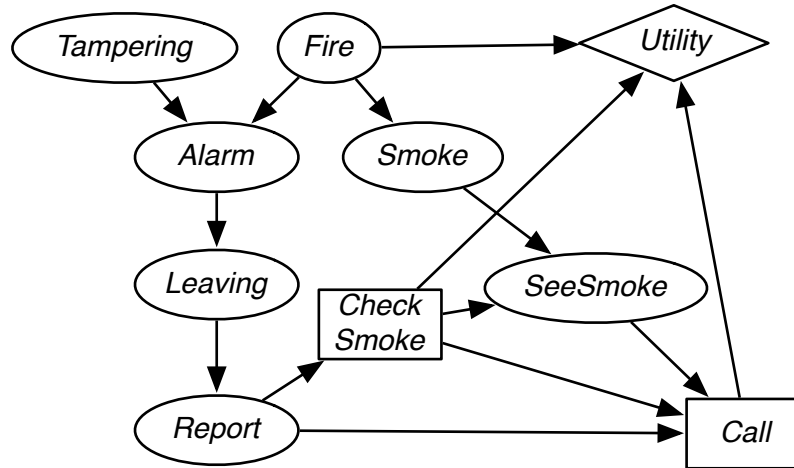


Figure 9.1: Fire Decision Network

```

115 f_ta = Prob(Ta,[],[0.98,0.02])
116 f-fi = Prob(Fi,[],[0.99,0.01])
117 f-sm = Prob(Sm,[Fi],[0.99,0.01,0.1,0.9])
118 f-al = Prob(Al,[Fi,Ta],[0.9999, 0.0001, 0.15, 0.85, 0.01, 0.99, 0.5, 0.5])
119 f-lv = Prob(Le,[Al],[0.999, 0.001, 0.12, 0.88])
120 f-re = Prob(Re,[Le],[0.99, 0.01, 0.25, 0.75])
121 f-ss = Prob(SS,[CS,Sm],[1,0,1,0,1,0,0,1])
122
123 ut = Utility([CS,Fi,Call],[0,-200,-5000,-200,-20,-220,-5020,-220])
124
125 dnf = DecisionNetwork([Ta,Fi,Al,Le,Sm,Call,SS,CS,Re],[f_ta,f-fi,f-sm,f-al,f-lv,f-re,f-ss,ut])
126 # v,p = VE_DN(dnf).optimize()
127 # for df in p: print(df,"\n")

```

The following is the representation of the cheating decision of Figure 9.2. Note that we keep the names of the variables short (less than 8 characters) so that tables Python prints look good.

```

129 grades = ["A","B","C","F"]
130 Wa = Variable("Watched", boolean)
131 CC1 = Variable("Caught1", boolean)
132 CC2 = Variable("Caught2", boolean)
133 Pun = Variable("Punish",["None","Suspension","Recorded"])
134 Gr1 = Variable("Grade_1",grades)
135 Gr2 = Variable("Grade_2",grades)
136 GrF = Variable("Fin_Gr",grades)
137 Ch1 = DecisionVariable("Cheat_1", boolean,set())
138 Ch2 = DecisionVariable("Cheat_2", boolean,{Ch1,CC1})
139
140 p_wa = Prob(Wa,[],[0.7, 0.3])

```

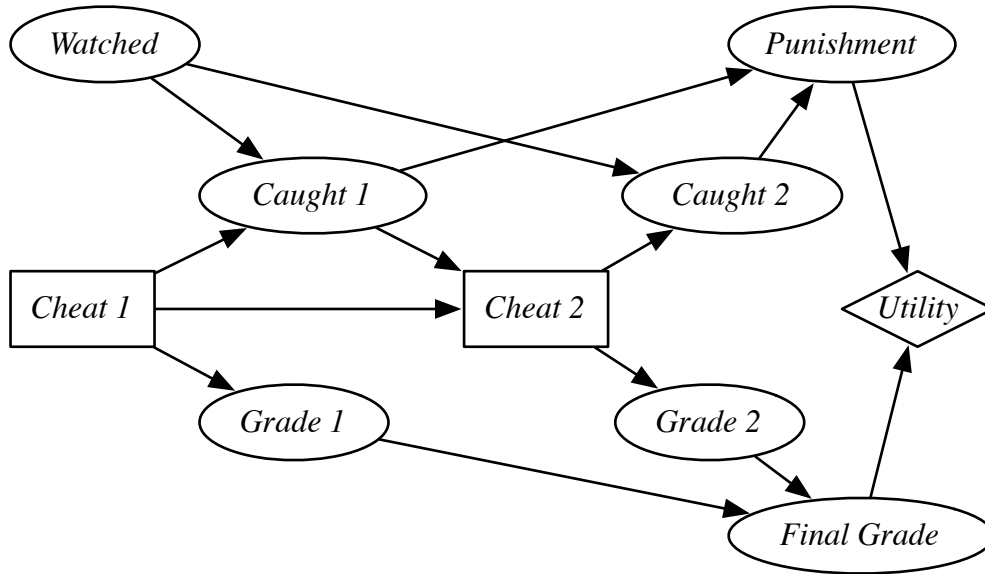


Figure 9.2: Cheating Decision Network

```

141 p_cc1 = Prob(CC1,[Wa,Ch1],[1.0, 0.0, 0.9, 0.1, 1.0, 0.0, 0.5, 0.5])
142 p_cc2 = Prob(CC2,[Wa,Ch2],[1.0, 0.0, 0.9, 0.1, 1.0, 0.0, 0.5, 0.5])
143 p_pun = Prob(Pun,[CC1,CC2],[1.0, 0.0, 0.0, 0.5, 0.4, 0.1, 0.6, 0.2, 0.2, 0.2, 0.5, 0.3])
144 p_gr1 = Prob(Gr1,[Ch1], [0.2, 0.3, 0.3, 0.2, 0.5, 0.3, 0.2, 0.0])
145 p_gr2 = Prob(Gr2,[Ch2], [0.2, 0.3, 0.3, 0.2, 0.5, 0.25, 0.25, 0.0])
146 p_fg = Prob(GrF,[Gr1,Gr2],
147             [1.0, 0.0, 0.0, 0.0, 0.5, 0.5, 0.0, 0.0, 0.25, 0.5, 0.25, 0.0, 0.25,
148              0.25, 0.25, 0.25, 0.5, 0.5, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.5,
149              0.5, 0.0, 0.0, 0.25, 0.5, 0.25, 0.25, 0.5, 0.25, 0.0, 0.0, 0.5, 0.5,
150              0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.25, 0.75, 0.25, 0.5, 0.25, 0.0,
151              0.0, 0.25, 0.5, 0.25, 0.0, 0.0, 0.25, 0.75, 0.0, 0.0, 0.0, 1.0])
152 utc = Utility([Pun,GrF],[100,90,70,50,40,20,10,0,70,60,40,20])
153
154 dnc = DecisionNetwork([Pun,CC2,Wa,GrF,Gr2,Gr1,Ch2,CC1,Ch1],
155                       [p_wa, p_cc1, p_cc2, p_pun, p_gr1, p_gr2,p_fg,utc])
156
157 # VE_DN.max_display_level = 3 # if you want to show lots of detail
158 # v,p = VE_DN(dnc).optimize(); print(v)
159 # for df in p: print(df,"\n") # print decision functions

```

## 9.2 Markov Decision Processes

We will represent a **Markov decision process (MDP)** directly, rather than using the variable elimination code, as we did for decision networks.

States and actions are represented as lists of strings. The data structures for transitions, rewards, q-values, etc., use the index of the state or the action. The

names of the state with index  $i$  is in `states[i]`, and the name of action with index  $i$  is in `actions[i]`.

```

mdpProblem.py — Representations for Markov Decision Processes
11 from utilities import argmax
12
13 class MDP(object):
14     def __init__(self, states, actions, trans, reward, discount):
15         """states is a list or tuple of states.
16            actions is a list or tuple of actions
17            trans[s][a][s'] represents P(s'|a,s)
18            reward[s][a] gives the expected reward of doing a in state s
19            discount is a real in the range [0,1]
20         """
21         self.states = states
22         self.actions = actions
23         self.trans = trans
24         self.reward = reward
25         self.discount = discount
26         self.v0 = [0 for s in states] # initial value function

```

2 state partying example:

```

mdpExamples.py — MDP Examples
11 from mdpProblem import MDP
12 ##### Partying Decision Example #####
13
14 # States: Healthy Sick
15 # Actions: Relax Party
16
17 # trans[s][a][s'] gives P(s'|a,s)
18 #           Relax      Party
19 trans2 = (((0.95,0.05), (0.7, 0.3)), # Healthy
20           ((0.5,0.5), (0.1, 0.9)) # Sick
21           )
22
23 # reward[s][a] gives the expected reward of doing a in state s.
24 reward2 = ((7,10),(0,2))
25
26 healthy2 = MDP(['Healthy','Sick'], ['Relax','Party'], trans2, reward2, discount=0.8)

```

Tiny game from Example 11.7 and Figure 11.8 of Poole and Mackworth, 2010:

```

mdpExamples.py — (continued)
28 ## Tiny Game from Example 11.7 and Figure 11.8 of Poole and Mackworth, 2010 #
29
30 # actions      up           right        upC        left
31 transt = (((0.1,0.1,0.8,0,0,0), (0,1,0,0,0,0), (0,0,1,0,0,0), (1,0,0,0,0,0)), #s0
32           ((0.1,0.1,0.8,0,0,0), (0,1,0,0,0,0), (0,0,0,1,0,0), (1,0,0,0,0,0)),
33           #s1

```

```

33         ((0,0,0.1,0.1,0.8,0), (0,0,0,1,0,0), (0,0,0,0,1,0), (0,0,1,0,0,0)),
#s2
34         ((0,0,0.1,0.1,0,0.8), (0,0,0,1,0,0), (0,0,0,0,0,1), (0,0,1,0,0,0)),
#s3
35         ((0.1,0,0,0,0.8,0.1), (0,0,0,0,0,1), (0,0,0,0,1,0), (1,0,0,0,0,0)),
#s4
36         ((0,0,0,0,0.1,0.9), (0,0,0,0,0,1), (0,0,0,0,0,1), (0,0,0,0,1,0)) ) #s5
37
38 # actions    up rt upC left
39 rewardt = ((-0.1, 0, -1, -1), #s0
40            (-0.1, -1, -2, 0), #s1
41            (-10, 0, -1, -100), #s2
42            (-0.1, -1, -1, 0), #s3
43            (-1, 0, -2, 10), #s4
44            (-1, -1, -2, 0)) #s5
45
46 mdpt = MDP(['s0','s1','s2','s3','s4','s5'], # states
47            ['up', 'right', 'upC', 'left'], # actions
48            transt, rewardt, discount=0.9)

```

## 9.3 Value Iteration

This implements value iteration, storing  $V$ .

This uses indexes of the states and actions (not the names). A value function is list,  $v$ , such that  $v[s]$  is the value for state with index  $s$ . Similarly a policy  $pi$  is represented as a list where  $pi[s]$ , where  $s$  is the index of a state, returns the index of the action.

```

mdpProblem.py — (continued)
28 def vi1(self,v):
29     """carry out one iteration of value iteration and
30     returns a value function (a list of a value for each state).
31     v is the previous value function.
32     """
33     return [max([self.reward[s][a]+self.discount*product(self.trans[s][a],v)
34                  for a in range(len(self.actions))])
35             for s in range(len(self.states))]
36
37 def vi(self,v0,n):
38     """carries out n iterations of value iteration starting with value v0.
39
40     Returns a value function
41     """
42     val = self.v0
43     for i in range(n):
44         val= self.vi1(val)
45     return val
46

```

```

47 def policy(self,v):
48     """returns an optimal policy assuming the next value function is v
49     v is a list of values for each state
50     returns a list of the indexes of optimal actions for each state
51     """
52     return [argmax(enumerate([self.reward[s][a]+self.discount*product(self.trans[s][a],v)
53                               for a in range(len(self.actions))]))
54             for s in range(len(self.states))]
55
56 def q(self,v):
57     """returns the one-step-lookahead q-value assuming the next value function is v
58     v is a list of values for each state
59     returns a list of q values for each state. so that q[s][a] represents Q(s,a)
60     """
61     return [[self.reward[s][a]+self.discount*product(self.trans[s][a],v)
62             for a in range(len(self.actions))]
63             for s in range(len(self.states))]

```

mdpProblem.py — (continued)

```

65 def product(l1,l2):
66     """returns the dot product of l1 and l2"""
67     return sum([i1*i2 for (i1,i2) in zip(l1,l2)])

```

The following gives a trace for the examples:

mdpExamples.py — (continued)

```

50 def trace(mdp,numiter):
51     print("Q values are shown as",[[st+"_"+ac for ac in mdp.actions] for st in mdp.states])
52     print("One step lookahead Q-values:")
53     print(mdp.q(mdp.v0))
54     print("Values are for the states:", mdp.states)
55     print("One step lookahead values:")
56     print(mdp.vi(mdp.v0,1))
57     print("Two step lookahead Q-values:")
58     print(mdp.q(mdp.vi(mdp.v0,1)))
59     print("Two step lookahead values:")
60     print(mdp.vi(mdp.v0,2))
61     vfin = mdp.vi(mdp.v0,numiter)
62     print("After",numiter,"iterations, values:")
63     print(vfin)
64     print("After",numiter,"iterations, Q-values:")
65     print(mdp.q(vfin))
66     print("After",numiter,"iterations, Policy:",
67           [st+"->" + mdp.actions[act] for (st,act) in zip(mdp.states ,mdp.policy(vfin))])
68
69 # Try the following:
70 # trace(healthy2,10)

```

**Exercise 9.1** Implement value iteration that stores the Q-values rather than the V-values. Does it work better than storing V? (What might better mean?)



**Exercise 9.2** Implement 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 change the most, and then update the previous ones, but that is not so straightforward to implement, because you need to find those previous states.



## Multiagent Systems

### 10.1 Minimax

Here we consider two-player zero-sum games. Here a player only wins when another player loses. This can be modeled as where there is a single utility which one agent (the maximizing agent) is trying minimize and the other agent (the minimizing agent) is trying to minimize.

#### 10.1.1 Creating a two-player game

```
masProblem.py — A Multiagent Problem
11 from utilities import Displayable
12
13 class Node(Displayable):
14     """A node in a search tree. It has a
15     name a string
16     isMax is True if it is a maximizing node, otherwise it is minimizing node
17     children is the list of children
18     value is what it evaluates to if it is a leaf.
19     """
20     def __init__(self, name, isMax, value, children):
21         self.name = name
22         self.isMax = isMax
23         self.value = value
24         self.allchildren = children
25
26     def isLeaf(self):
27         """returns true if this is a leaf node"""
28         return self.allchildren is None
29
```

```

30     def children(self):
31         """returns the list of all children."""
32         return self.allchildren
33
34     def evaluate(self):
35         """returns the evaluation for this node if it is a leaf"""
36         return self.value

```

The following gives the tree from Figure 10.5 of the book. Note how 888 is used as a value here, but never appears in the trace.

```

masProblem.py — (continued)
38 fig10_5 = Node("a", True, None, [
39     Node("b", False, None, [
40         Node("d", True, None, [
41             Node("h", False, None, [
42                 Node("h1", True, 7, None),
43                 Node("h2", True, 9, None)]],
44             Node("i", False, None, [
45                 Node("i1", True, 6, None),
46                 Node("i2", True, 888, None)]])],
47         Node("e", True, None, [
48             Node("j", False, None, [
49                 Node("j1", True, 11, None),
50                 Node("j2", True, 12, None)]],
51             Node("k", False, None, [
52                 Node("k1", True, 888, None),
53                 Node("k2", True, 888, None)]])]),
54     Node("c", False, None, [
55         Node("f", True, None, [
56             Node("l", False, None, [
57                 Node("l1", True, 5, None),
58                 Node("l2", True, 888, None)]],
59             Node("m", False, None, [
60                 Node("m1", True, 4, None),
61                 Node("m2", True, 888, None)]])],
62     Node("g", True, None, [
63         Node("n", False, None, [
64             Node("n1", True, 888, None),
65             Node("n2", True, 888, None)]],
66         Node("o", False, None, [
67             Node("o1", True, 888, None),
68             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 10.1); 3 numbers that add to 15 correspond exactly to the winning positions of tic-tac-toe played on the magic square.

Note that we do not remove symmetries. (What are the symmetries? How

6	1	8
7	5	3
2	9	4

Figure 10.1: Magic Square

do the symmetries of tic-tac-toe translate here?)

masProblem.py — (continued)

```

70
71 class Magic_sum(Node):
72     def __init__(self, xmove=True, last_move=None,
73                 available=[1,2,3,4,5,6,7,8,9], x=[], o=[]):
74         """This is a node in the search for the magic-sum game.
75         xmove is True if the next move belongs to X.
76         last_move is the number selected in the last move
77         available is the list of numbers that are available to be chosen
78         x is the list of numbers already chosen by x
79         o is the list of numbers already chosen by o
80         """
81         self.isMax = self.xmove = xmove
82         self.last_move = last_move
83         self.available = available
84         self.x = x
85         self.o = o
86         self.allchildren = None #computed on demand
87         lm = str(last_move)
88         self.name = "start" if not last_move else "o="+lm if xmove else "x="+lm
89
90     def children(self):
91         if self.allchildren is None:
92             if self.xmove:
93                 self.allchildren = [
94                     Magic_sum(xmove = not self.xmove,
95                             last_move = sel,
96                             available = [e for e in self.available if e is not sel],
97                             x = self.x+[sel],
98                             o = self.o)
99                     for sel in self.available]
100             else:
101                 self.allchildren = [
102                     Magic_sum(xmove = not self.xmove,
103                             last_move = sel,
104                             available = [e for e in self.available if e is not sel],
105                             x = self.x,
106                             o = self.o+[sel])
107                     for sel in self.available]
108         return self.allchildren
109

```

```

110 def isLeaf(self):
111     """A leaf has no numbers available or is a win for one of the players.
112     We only need to check for a win for o if it is currently x's turn,
113     and only check for a win for x if it is o's turn (otherwise it would
114     have been a win earlier).
115     """
116     return (self.available == [] or
117            (sum_to_15(self.last_move,self.o)
118             if self.xmove
119             else sum_to_15(self.last_move,self.x)))
120
121 def evaluate(self):
122     if self.xmove and sum_to_15(self.last_move,self.o):
123         return -1
124     elif not self.xmove and sum_to_15(self.last_move,self.x):
125         return 1
126     else:
127         return 0
128
129 def sum_to_15(last,selected):
130     """is true if last, together with two other elements of selected sum to 15.
131     """
132     return any(last+a+b == 15
133               for a in selected if a != last
134               for b in selected if b != last and b != a)

```

### 10.1.2 Minimax and $\alpha$ - $\beta$ Pruning

This is a naive depth-first **minimax algorithm**:

```

_____masMiniMax.py — Minimax search with alpha-beta pruning_____
11 def minimax(node):
12     """returns the value of node, and a best path for the agents
13     """
14     if node.isLeaf():
15         return node.evaluate(),None
16     elif node.isMax:
17         max_score = -999
18         max_path = None
19         for C in node.children():
20             score,path = minimax(C,depth+1)
21             if score > max_score:
22                 max_score = score
23                 max_path = C.name,path
24         return max_score,max_path
25     else:
26         min_score = 999
27         min_path = None
28         for C in node.children():
29             score,path = minimax(C,depth+1)
30             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  $\alpha$ - $\beta$  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, alpha and beta are cutoffs, depth is the depth
37     returns value, path
38     where path is a sequence of nodes that results in the value"""
39     node.display(2," *depth,"minimax_alpha_beta(",node.name," ",alpha," ",beta,"")
40     best=None # only used if it will be pruned
41     if node.isLeaf():
42         node.display(2," *depth,"returning leaf value",node.evaluate())
43         return node.evaluate(),None
44     elif node.isMax:
45         for C in node.children():
46             score,path = minimax_alpha_beta(C,alpha,beta,depth+1)
47             if score >= beta: # beta pruning
48                 node.display(2," *depth,"pruned due to beta=",beta,"C=",C.name)
49                 return score, None
50             if score > alpha:
51                 alpha = score
52                 best = C.name, path
53             node.display(2," *depth,"returning max alpha",alpha,"best",best)
54             return alpha,best
55     else:
56         for C in node.children():
57             score,path = minimax_alpha_beta(C,alpha,beta,depth+1)
58             if score <= alpha: # alpha pruning
59                 node.display(2," *depth,"pruned due to alpha=",alpha,"C=",C.name)
60                 return score, None
61             if score < beta:
62                 beta=score
63                 best = C.name,path
64             node.display(2," *depth,"returning min beta",beta,"best=",best)
65             return beta,best

```

Testing:

```

masMiniMax.py — (continued)
67 from masProblem import fig10_5, Magic_sum, Node
68
69 # Node.max_display_level=2 # print detailed trace
70 # minimax_alpha_beta(fig10_5, -9999, 9999,0)
71 # minimax_alpha_beta(Magic_sum(), -9999, 9999,0)
72
73 #To see how much time alpha-beta pruning can save over minimax, uncomment the following:
74 ## import timeit
75 ## timeit.Timer("minimax(Magic_sum())",setup="from __main__ import minimax, Magic_sum"

```

```
76 | ##                ).timeit(number=1)
77 | ## trace=False
78 | ## timeit.Timer("minimax_alpha_beta(Magic_sum(), -9999, 9999,0)",
79 | ##              setup="from __main__ import minimax_alpha_beta, Magic_sum"
80 | ##              ).timeit(number=1)
```



## Reinforcement Learning

### 11.1 Representing Agents and Environments

When the learning agent does an action in the environment, it observes a  $(state, reward)$  pair from the environment. The *state* is the world state; this is the fully observable assumption.

An RL environment implements a  $do(action)$  method that returns a  $(state, reward)$  pair.

```
_____rlProblem.py — Representations for Reinforcement Learning_____
11 import random
12 from utilities import Displayable, flip
13
14 class RL_env(Displayable):
15     def __init__(self, actions, state):
16         self.actions = actions # set of actions
17         self.state = state     # initial state
18
19     def do(self, action):
20         """do action
21         returns state, reward
22         """
23         raise NotImplementedError("RL_env.do") # abstract method
```

Here is the definition of the simple 2-state, 2-action party/relax decision.

```
_____rlProblem.py — (continued)_____
25 class Healthy_env(RL_env):
26     def __init__(self):
27         RL_env.__init__(self, ["party", "relax"], "healthy")
28
29     def do(self, action):
```

```

30     """updates the state based on the agent doing action.
31     returns state,reward
32     """
33     if self.state=="healthy":
34         if action=="party":
35             self.state = "healthy" if flip(0.7) else "sick"
36             reward = 10
37         else: # action=="relax"
38             self.state = "healthy" if flip(0.95) else "sick"
39             reward = 7
40     else: # self.state=="sick"
41         if action=="party":
42             self.state = "healthy" if flip(0.1) else "sick"
43             reward = 2
44         else:
45             self.state = "healthy" if flip(0.5) else "sick"
46             reward = 0
47     return self.state,reward

```

### 11.1.1 Simulating an environment from an MDP

Given the definition for an MDP (page 173), *Env\_from\_MDP* takes in an MDP and simulates the environment with those dynamics.

Note that the MDP does not contain enough information to simulate a system, because it loses any dependency between the rewards and the resulting state; here we assume the agent always received the average reward for the state and action.

```

rlProblem.py — (continued)
49 class Env_from_MDP(RL_env):
50     def __init__(self, mdp):
51         initial_state = mdp.states[0]
52         RL_env.__init__(self,mdp.actions, initial_state)
53         self.mdp = mdp
54         self.action_index = {action:index for (index,action) in enumerate(mdp.actions)}
55         self.state_index = {state:index for (index,state) in enumerate(mdp.states)}
56
57     def do(self, action):
58         """updates the state based on the agent doing action.
59         returns state,reward
60         """
61         action_ind = self.action_index[action]
62         state_ind = self.state_index[self.state]
63         self.state = pick_from_dist(self.mdp.trans[state_ind][action_ind], self.mdp.states)
64         reward = self.mdp.reward[state_ind][action_ind]
65         return self.state, reward
66
67     def pick_from_dist(dist,values):
68         """

```

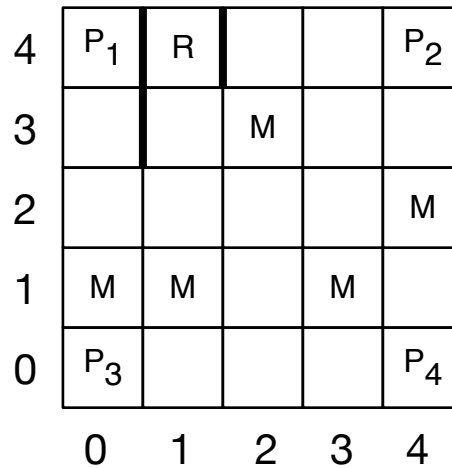


Figure 11.1: Monster game

```

69     e.g. pick_from_dist([0.3,0.5,0.2],['a','b','c']) should pick 'a' with probability 0.3, etc.
70     """
71     ran = random.random()
72     i=0
73     while ran>dist[i]:
74         ran -= dist[i]
75         i += 1
76     return values[i]

```

### 11.1.2 Simple Game

This is for the game depicted in Figure 11.1.

```

_____rlSimpleEnv.py — Simple game_____
11 import random
12 from utilities import flip
13 from rlProblem import RL_env
14
15 class Simple_game_env(RL_env):
16     xdim = 5
17     ydim = 5
18
19     vwalls = [(0,3), (0,4), (1,4)] # vertical walls right of these locations
20     hwalls = [] # not implemented
21     crashed_reward = -1
22
23     prize_locs = [(0,0), (0,4), (4,0), (4,4)]
24     prize_appears_prob = 0.3
25     prize_reward = 10
26

```

```

27 monster_locs = [(0,1), (1,1), (2,3), (3,1), (4,2)]
28 monster_appears_prob = 0.4
29 monster_reward_when_damaged = -10
30 repair_stations = [(1,4)]
31
32 actions = ["up","down","left","right"]
33
34 def __init__(self):
35     # State:
36     self.x = 2
37     self.y = 2
38     self.damaged = False
39     self.prize = None
40     # Statistics
41     self.number_steps = 0
42     self.total_reward = 0
43     self.min_reward = 0
44     self.min_step = 0
45     self.zero_crossing = 0
46     RL_env.__init__(self, Simple_game_env.actions,
47                     (self.x, self.y, self.damaged, self.prize))
48     self.display(2,"","Step","Tot Rew","Ave Rew",sep="\t")
49
50 def do(self,action):
51     """updates the state based on the agent doing action.
52     returns state,reward
53     """
54     reward = 0.0
55     # A prize can appear:
56     if self.prize is None and flip(self.prize_appears_prob):
57         self.prize = random.choice(self.prize_locs)
58     # Actions can be noisy
59     if flip(0.4):
60         actual_direction = random.choice(self.actions)
61     else:
62         actual_direction = action
63     # Modeling the actions given the actual direction
64     if actual_direction == "right":
65         if self.x==self.xdim-1 or (self.x,self.y) in self.vwalls:
66             reward += self.crashed_reward
67         else:
68             self.x += 1
69     elif actual_direction == "left":
70         if self.x==0 or (self.x-1,self.y) in self.vwalls:
71             reward += self.crashed_reward
72         else:
73             self.x += -1
74     elif actual_direction == "up":
75         if self.y==self.ydim-1:
76             reward += self.crashed_reward

```

```

77         else:
78             self.y += 1
79     elif actual_direction == "down":
80         if self.y==0:
81             reward += self.crashed_reward
82         else:
83             self.y += -1
84     else:
85         raise RuntimeError("unknown_direction "+str(direction))
86
87     # Monsters
88     if (self.x,self.y) in self.monster_locs and flip(self.monster_appears_prob):
89         if self.damaged:
90             reward += self.monster_reward_when_damaged
91         else:
92             self.damaged = True
93     if (self.x,self.y) in self.repair_stations:
94         self.damaged = False
95
96     # Prizes
97     if (self.x,self.y) == self.prize:
98         reward += self.prize_reward
99         self.prize = None
100
101     # Statistics
102     self.number_steps += 1
103     self.total_reward += reward
104     if self.total_reward < self.min_reward:
105         self.min_reward = self.total_reward
106         self.min_step = self.number_steps
107     if self.total_reward>0 and reward>self.total_reward:
108         self.zero_crossing = self.number_steps
109     self.display(2,"",self.number_steps,self.total_reward,
110                 self.total_reward/self.number_steps,sep="\t")
111
112     return (self.x, self.y, self.damaged, self.prize), reward

```

### 11.1.3 Evaluation and Plotting

```

r1Plot.py — RL Plotter
11 import matplotlib.pyplot as plt
12
13 def plot_rl(ag, label=None, yplot='Total', step_size=None,
14            steps_explore=1000, steps_exploit=1000, xscale='linear'):
15     """
16     plots the agent ag
17     label is the label for the plot
18     yplot is 'Average' or 'Total'
19     step_size is the number of steps between each point plotted

```

```

20     steps_explore is the number of steps the agent spends exploring
21     steps_exploit is the number of steps the agent spends exploiting
22     xscale is 'log' or 'linear'
23
24     returns total reward when exploring, total reward when exploiting
25     """
26     assert yplot in ['Average', 'Total']
27     if step_size is None:
28         step_size = max(1, (steps_explore+steps_exploit)//500)
29     if label is None:
30         label = ag.label
31     ag.max_display_level, old_md1 = 1, ag.max_display_level
32     plt.ion()
33     plt.xscale(xscale)
34     plt.xlabel("step")
35     plt.ylabel(yplot+" reward")
36     steps = []      # steps
37     rewards = []    # return
38     ag.restart()
39     step = 0
40     while step < steps_explore:
41         ag.do(step_size)
42         step += step_size
43         steps.append(step)
44         if yplot == "Average":
45             rewards.append(ag.acc_rewards/step)
46         else:
47             rewards.append(ag.acc_rewards)
48     acc_rewards_exploring = ag.acc_rewards
49     ag.explore, explore_save = 0, ag.explore
50     while step < steps_explore+steps_exploit:
51         ag.do(step_size)
52         step += step_size
53         steps.append(step)
54         if yplot == "Average":
55             rewards.append(ag.acc_rewards/step)
56         else:
57             rewards.append(ag.acc_rewards)
58     plt.plot(steps, rewards, label=label)
59     plt.legend(loc="upper left")
60     plt.draw()
61     ag.max_display_level = old_md1
62     ag.explore=explore_save
63     return acc_rewards_exploring, ag.acc_rewards-acc_rewards_exploring

```

## 11.2 Q Learning

To run the Q-learning demo, in folder “aipython”, load “rlQTest.py”, and copy and paste the example queries at the bottom of that file. This assumes Python 3.

```

rlQLearner.py — Q Learning
11 import random
12 from utilities import Displayable, argmax, flip
13
14 class RL_agent(Displayable):
15     """An RL_Agent
16     has percepts (s, r) for some state s and real reward r
17     """

rlQLearner.py — (continued)
19 class Q_learner(RL_agent):
20     """A Q-learning agent has
21     belief-state consisting of
22     state is the previous state
23     q is a {(state,action):value} dict
24     visits is a {(state,action):n} dict. n is how many times action was done in state
25     acc_rewards is the accumulated reward
26
27     it observes (s, r) for some world-state s and real reward r
28     """

rlQLearner.py — (continued)
30 def __init__(self, env, discount, explore=0.1, fixed_alpha=True, alpha=0.2,
31             alpha_fun=lambda k:1/k,
32             qinit=0, label="Q_learner"):
33     """env is the environment to interact with.
34     discount is the discount factor
35     explore is the proportion of time the agent will explore
36     fixed_alpha specifies whether alpha is fixed or varies with the number of visits
37     alpha is the weight of new experiences compared to old experiences
38     alpha_fun is a function that computes alpha from the number of visits
39     qinit is the initial value of the Q's
40     label is the label for plotting
41     """
42     RL_agent.__init__(self)
43     self.env = env
44     self.actions = env.actions
45     self.discount = discount
46     self.explore = explore
47     self.fixed_alpha = fixed_alpha
48     self.alpha = alpha
49     self.alpha_fun = alpha_fun

```





**Exercise 11.1** 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, soft-max and optimism in the face of uncertainty.

**Exercise 11.2** Implement SARSA. Hint: it does not do a *max* in *do*. Instead it needs to choose *next\_act* before it does the update.

### 11.2.1 Testing Q-learning

The first tests are for the 2-action 2-state

```

rlQTest.py — RL Q Tester
11 from rlProblem import Healthy_env
12 from rlQLearner import Q_learner
13 from rlPlot import plot_rl
14
15 env = Healthy_env()
16 ag = Q_learner(env, 0.7)
17 ag_opt = Q_learner(env, 0.7, qinit=100, label="optimistic" ) # optimistic agent
18 ag_exp_l = Q_learner(env, 0.7, explore=0.01, label="less explore")
19 ag_exp_m = Q_learner(env, 0.7, explore=0.5, label="more explore")
20 ag_disc = Q_learner(env, 0.9, qinit=100, label="disc 0.9")
21 ag_va = Q_learner(env, 0.7, qinit=100, fixed_alpha=False, alpha_fun=lambda k:10/(9+k), label="alpha=10/(9+k)")
22
23 # ag.max_display_level = 2
24 # ag.do(20)
25 # ag.q # get the learned q-values
26 # ag.max_display_level = 1
27 # ag.do(1000)
28 # ag.q # get the learned q-values
29 # plot_rl(ag, yplot="Average")
30 # plot_rl(ag_opt, yplot="Average")
31 # plot_rl(ag_exp_l, yplot="Average")
32 # plot_rl(ag_exp_m, yplot="Average")
33 # plot_rl(ag_disc, yplot="Average")
34 # plot_rl(ag_va, yplot="Average")
35
36 from mdpExamples import mdpt
37 from rlProblem import Env_from_MDP
38 envt = Env_from_MDP(mdpt)
39 agt = Q_learner(envt, 0.8)
40 # agt.do(20)
41
42 from rlSimpleEnv import Simple_game_env
43 senv = Simple_game_env()
44 sag1 = Q_learner(senv, 0.9, explore=0.2, fixed_alpha=True, alpha=0.1)
45 # plot_rl(sag1, steps_explore=100000, steps_exploit=100000, label="alpha="+str(sag1.alpha))
46 sag2 = Q_learner(senv, 0.9, explore=0.2, fixed_alpha=False)
47 # plot_rl(sag2, steps_explore=100000, steps_exploit=100000, label="alpha=1/k")
48 sag3 = Q_learner(senv, 0.9, explore=0.2, fixed_alpha=False, alpha_fun=lambda k:10/(9+k))
49 # plot_rl(sag3, steps_explore=100000, steps_exploit=100000, label="alpha=10/(9+k)")

```

## 11.3 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.

The model-based reinforcement learner used the following data structures:

- $q[s, a]$  is dictionary that, given a  $(s, a)$  pair returns the  $Q$ -value, the estimate of the future (discounted) value of being in state  $s$  and doing action  $a$ .
- $r[s, a]$  is dictionary that, given a  $(s, a)$  pair returns the average reward from doing  $a$  in state  $s$ .
- $t[s, a, s']$  is dictionary that, given a  $(s, a, s')$  tuple returns the number of times  $a$  was done in state  $s$ , with the result being state  $s'$ .
- $visits[s, a]$  is dictionary that, given a  $(s, a)$  pair returns the number of times action  $a$  was carried out in state  $s$ .
- $res\_states[s, a]$  is dictionary that, given a  $(s, a)$  pair returns the list of resulting states that have occurred when action  $a$  was carried out in state  $s$ . This is used in the asynchronous value iteration to determine the  $s'$  states to sum over.
- $visits\_list$  is a list of  $(s, a)$  pair that have been carried out. This is used to ensure there is no divide-by zero in the asynchronous value iteration. Note that this could be constructed from  $r$ ,  $visits$  or  $res\_states$  by enumerating the keys, but needs to be a list for *random.choice*, and we don't want to keep recreating it.

```

rlModelLearner.py — Model-based Reinforcement Learner
11 import random
12 from rlQLearner import RL_agent
13 from utilities import Displayable, argmax, flip
14
15 class Model_based_reinforcement_learner(RL_agent):
16     """A Model-based reinforcement learner
17     """
18
19     def __init__(self, env, discount, explore=0.1, qinit=0,
20                 updates_per_step=10, label="MBR_learner"):
21         """env is the environment to interact with.
22         discount is the discount factor
23         explore is the proportion of time the agent will explore
24         qinit is the initial value of the Q's
25         updates_per_step is the number of AVI updates per action
26         label is the label for plotting

```

```

27         """
28         RL_agent.__init__(self)
29         self.env = env
30         self.actions = env.actions
31         self.discount = discount
32         self.explore = explore
33         self.qinit = qinit
34         self.updates_per_step = updates_per_step
35         self.label = label
36         self.restart()

```

---

rlModelLearner.py — (continued)

---

```

38     def restart(self):
39         """make the agent relearn, and reset the accumulated rewards
40         """
41         self.acc_rewards = 0
42         self.state = self.env.state
43         self.q = {} # {(st,action):q_value} map
44         self.r = {} # {(st,action):reward} map
45         self.t = {} # {(st,action,st_next):count} map
46         self.visits = {} # {(st,action):count} map
47         self.res_states = {} # {(st,action):set_of_states} map
48         self.visits_list = [] # list of (st,action)
49         self.previous_action = None

```

---

rlModelLearner.py — (continued)

---

```

51     def do(self,num_steps=100):
52         """do num_steps of interaction with the environment
53         for each action, do updates_per_step iterations of asynchronous value iteration
54         """
55         for step in range(num_steps):
56             pst = self.state # previous state
57             action = self.select_action(pst)
58             self.state,reward = self.env.do(action)
59             self.acc_rewards += reward
60             self.t[(pst,action,self.state)] = self.t.get((pst, action,self.state),0)+1
61             if (pst,action) in self.visits:
62                 self.visits[(pst,action)] += 1
63                 self.r[(pst,action)] += (reward-self.r[(pst,action)])/self.visits[(pst,action)]
64                 self.res_states[(pst,action)].add(self.state)
65             else:
66                 self.visits[(pst,action)] = 1
67                 self.r[(pst,action)] = reward
68                 self.res_states[(pst,action)] = {self.state}
69                 self.visits_list.append((pst,action))
70             st,act = pst,action #initial state-action pair for AVI
71             for update in range(self.updates_per_step):
72                 self.q[(st,act)] = self.r[(st,act)]+self.discount*(
73                     sum(self.t[st,act,rst]/self.visits[st,act]*
74                        max(self.q.get((rst,nact),self.qinit) for nact in self.actions)

```

```

75         for rst in self.res_states[(st,act)]))
76         st,act = random.choice(self.visits_list)

-----rlModelLearner.py --- (continued) -----
78     def select_action(self, state):
79         """returns an action to carry out for the current agent
80         given the state, and the q-function
81         """
82         if flip(self.explore):
83             return random.choice(self.actions)
84         else:
85             return argmax((next_act, self.q.get((state, next_act),self.qinit))
86                           for next_act in self.actions)

-----rlModelLearner.py --- (continued) -----
88 from rlQTest import env # simple game environment
89 mb11 = Model_based_reinforcement_learner(env,0.9,updates_per_step=10)
90 # plot_rl(mb11,steps_explore=100000,steps_exploit=100000,label="model-based(10)")
91 mb12 = Model_based_reinforcement_learner(env,0.9,updates_per_step=1)
92 # plot_rl(mb12,steps_explore=100000,steps_exploit=100000,label="model-based(1)")

```

**Exercise 11.3** If there was only one update per step, the algorithm can 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 11.4** It is possible to implement the model-based reinforcement learner by replacing  $q$ ,  $r$ ,  $visits$ ,  $res\_states$  with a single dictionary that returns a tuple  $(q, r, v, tm)$  where  $q$ ,  $r$  and  $v$  are numbers, and  $tm$  is a map from resulting states into counts. Does this make the algorithm easier to understand? Does this make the algorithm more efficient?

**Exercise 11.5** If the states and the actions were mapped into integers, the dictionaries could be implemented more efficiently as arrays. This entails an extra step in specifying problems. Implement this for the simple game. Is it more efficient?

## 11.4 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.

### 11.4.1 Representing Features

A feature is a function from state and action. To construct the features for a domain, we construct a function that takes a state and an action and returns the list of all feature values for that state and action. This feature set is redesigned for each problem.

`get_features(state, action)` returns the feature values appropriate for the simple game.

```

rlSimpleGameFeatures.py — Feature-based Reinforcement Learner
11 from rlSimpleEnv import Simple_game_env
12 from rlProblem import RL_env
13
14 def get_features(state, action):
15     """returns the list of feature values for the state-action pair
16     """
17     assert action in Simple_game_env.actions
18     (x,y,d,p) = state
19     # f1: would go to a monster
20     f1 = monster_ahead(x,y,action)
21     # f2: would crash into wall
22     f2 = wall_ahead(x,y,action)
23     # f3: action is towards a prize
24     f3 = towards_prize(x,y,action,p)
25     # f4: damaged and action is toward repair station
26     f4 = towards_repair(x,y,action) if d else 0
27     # f5: damaged and towards monster
28     f5 = 1 if d and f1 else 0
29     # f6: damaged
30     f6 = 1 if d else 0
31     # f7: not damaged
32     f7 = 1-f6
33     # f8: damaged and prize ahead
34     f8 = 1 if d and f3 else 0
35     # f9: not damaged and prize ahead
36     f9 = 1 if not d and f3 else 0
37     features = [1,f1,f2,f3,f4,f5,f6,f7,f8,f9]
38     for pr in Simple_game_env.prize_locs+[None]:
39         if p==pr:
40             features += [x, 4-x, y, 4-y]
41         else:
42             features += [0, 0, 0, 0]
43     # fp04 feature for y when prize is at 0,4
44     # this knows about the wall to the right of the prize
45     if p==(0,4):
46         if x==0:
47             fp04 = y
48         elif y<3:
49             fp04 = y
50         else:
51             fp04 = 4-y
52     else:
53         fp04 = 0
54     features.append(fp04)
55     return features
56
57 def monster_ahead(x,y,action):

```

```

58     """returns 1 if the location expected to get to by doing
59     action from (x,y) can contain a monster.
60     """
61     if action == "right" and (x+1,y) in Simple_game_env.monster_locs:
62         return 1
63     elif action == "left" and (x-1,y) in Simple_game_env.monster_locs:
64         return 1
65     elif action == "up" and (x,y+1) in Simple_game_env.monster_locs:
66         return 1
67     elif action == "down" and (x,y-1) in Simple_game_env.monster_locs:
68         return 1
69     else:
70         return 0
71
72 def wall_ahead(x,y,action):
73     """returns 1 if there is a wall in the direction of action from (x,y).
74     This is complicated by the internal walls.
75     """
76     if action == "right" and (x==Simple_game_env.xdim-1 or (x,y) in Simple_game_env.vwalls):
77         return 1
78     elif action == "left" and (x==0 or (x-1,y) in Simple_game_env.vwalls):
79         return 1
80     elif action == "up" and y==Simple_game_env.ydim-1:
81         return 1
82     elif action == "down" and y==0:
83         return 1
84     else:
85         return 0
86
87 def towards_prize(x,y,action,p):
88     """action goes in the direction of the prize from (x,y)"""
89     if p is None:
90         return 0
91     elif p==(0,4): # take into account the wall near the top-left prize
92         if action == "left" and (x>1 or x==1 and y<3):
93             return 1
94         elif action == "down" and (x>0 and y>2):
95             return 1
96         elif action == "up" and (x==0 or y<2):
97             return 1
98         else:
99             return 0
100     else:
101         px,py = p
102         if p==(4,4) and x==0:
103             if (action=="right" and y<3) or (action=="down" and y>2) or (action=="up" and y<2):
104                 return 1
105             else:
106                 return 0
107         if (action == "up" and y<py) or (action == "down" and py<y):

```

```

108         return 1
109     elif (action == "left" and px<x) or (action == "right" and x<px):
110         return 1
111     else:
112         return 0
113
114 def towards_repair(x,y,action):
115     """returns 1 if action is towards the repair station.
116     """
117     if action == "up" and (x>0 and y<4 or x==0 and y<2):
118         return 1
119     elif action == "left" and x>1:
120         return 1
121     elif action == "right" and x==0 and y<3:
122         return 1
123     elif action == "down" and x==0 and y>2:
124         return 1
125     else:
126         return 0
127
128 def simp_features(state,action):
129     """returns a list of feature values for the state-action pair
130     """
131     assert action in Simple_game_env.actions
132     (x,y,d,p) = state
133     # f1: would go to a monster
134     f1 = monster_ahead(x,y,action)
135     # f2: would crash into wall
136     f2 = wall_ahead(x,y,action)
137     # f3: action is towards a prize
138     f3 = towards_prize(x,y,action,p)
139     return [1,f1,f2,f3]

```

### 11.4.2 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_____
11 import random
12 from rlQLearner import RL_agent
13 from utilities import Displayable, argmax, flip
14
15 class SARSA_LFA_learner(RL_agent):
16     """A SARSA_LFA learning agent has
17     belief-state consisting of
18     state is the previous state
19     q is a {(state,action):value} dict
20     visits is a {(state,action):n} dict. n is how many times action was done in state

```

```

21         acc_rewards is the accumulated reward
22
23     it observes (s, r) for some world-state s and real reward r
24     """
25     def __init__(self, env, get_features, discount, explore=0.2, step_size=0.01,
26                 winit=0, label="SARSA_LFA"):
27         """env is the feature environment to interact with
28         get_features is a function get_features(state,action) that returns the list of feature values
29         discount is the discount factor
30         explore is the proportion of time the agent will explore
31         step_size is gradient descent step size
32         winit is the initial value of the weights
33         label is the label for plotting
34         """
35         RL_agent.__init__(self)
36         self.env = env
37         self.get_features = get_features
38         self.actions = env.actions
39         self.discount = discount
40         self.explore = explore
41         self.step_size = step_size
42         self.winit = winit
43         self.label = label
44         self.restart()

```

`restart()` is used to make the learner relearn everything. This is used by the plotter to create new plots.

rlFeatures.py — (continued)

```

46     def restart(self):
47         """make the agent relearn, and reset the accumulated rewards
48         """
49         self.acc_rewards = 0
50         self.state = self.env.state
51         self.features = self.get_features(self.state, list(self.env.actions)[0])
52         self.weights = [self.winit for f in self.features]
53         self.action = self.select_action(self.state)

```

`do` takes in the number of steps.

rlFeatures.py — (continued)

```

55     def do(self, num_steps=100):
56         """do num_steps of interaction with the environment"""
57         self.display(2, "s\t a\t r\t ts'\t Q\t delta")
58         for i in range(num_steps):
59             next_state, reward = self.env.do(self.action)
60             self.acc_rewards += reward
61             next_action = self.select_action(next_state)
62             feature_values = self.get_features(self.state, self.action)
63             oldQ = dot_product(self.weights, feature_values)
64             nextQ = dot_product(self.weights, self.get_features(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                     dot_product(self.weights, feature_values), delta, sep='\t')
70         self.state = next_state
71         self.action = next_action
72
73     def select_action(self, state):
74         """returns an action to carry out for the current agent
75         given the state, and the q-function.
76         This implements an epsilon-greedy approach
77         where self.explore is the probability of exploring.
78         """
79         if flip(self.explore):
80             return random.choice(self.actions)
81         else:
82             return argmax((next_act, dot_product(self.weights,
83                                                  self.get_features(state, next_act)))
84                           for next_act in self.actions)
85
86     def show_actions(self, state=None):
87         """prints the value for each action in a state.
88         This may be useful for debugging.
89         """
90         if state is None:
91             state = self.state
92         for next_act in self.actions:
93             print(next_act, dot_product(self.weights, self.get_features(state, next_act)))
94
95     def dot_product(l1, l2):
96         return sum(e1*e2 for (e1, e2) in zip(l1, l2))

```

Test code:

```

_____rlFeatures.py — (continued)_____
99 from rlQTest import env # simple game environment
100 from rlSimpleGameFeatures import get_features, simp_features
101 from rlPlot import plot_rl
102
103 fa1 = SARSA_LFA_learner(env, get_features, 0.9, step_size=0.01)
104 #fa1.max_display_level = 2
105 #fa1.do(20)
106 #plot_rl(fa1, steps_explore=10000, steps_exploit=10000, label="SARSA_LFA(0.01)")
107 fas1 = SARSA_LFA_learner(env, simp_features, 0.9, step_size=0.01)
108 #plot_rl(fas1, steps_explore=10000, steps_exploit=10000, label="SARSA_LFA(simp)")

```

**Exercise 11.6** How does the step-size affect performance? Try different step sizes (e.g., 0.1, 0.001, other sizes in between). Explain the behaviour you observe. Which step size works best for this example. Explain what evidence you are basing your prediction on.

**Exercise 11.7** 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 11.8** For each of the following first predict, then plot, then explain the behaviour 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.

## 11.5 Learning to coordinate - UNFINISHED!!!!

Coordinating agents should implement the agent architecture. However, in that architecture, an agent calls the environment. That architecture was chosen because it was simple. However, it does not really work when there are multiple agents. In such cases, a coroutining architecture is more appropriate.

We assume there is an x-player, and a y-player.  $game[xa][ya][ag]$  gives value to the agent  $ag$  ( $ag$ =for the x-player) of the strategy of the x-agent doing  $xa$  and the y-agent doing  $ya$ .

```

learnCoordinate.py — Learning to Coordinate
11 from learnProblem import Learner
12
13 soccer = [[(-0.6,0.6),(-0.3,0.3)],[(-0.2,0.2),(-0.9,0.9)]]
14 football = [[(2,1),(0,0)],[(0,0),(1,2)]]
15 prisoners_game = [[(100,100),(0,1100)],[(1100,0),(1000,1000)]]
16
17 class Policy_hill_climbing(Learner):
18     def __init__(self,game)
```

## Relational Learning

### 12.1 Collaborative Filtering

Based on gradient descent algorithm of Koren, Y., Bell, R. and Volinsky, C., Matrix Factorization Techniques for Recommender Systems, IEEE Computer 2009.

This assumes the form of the dataset from movielens (<http://grouplens.org/datasets/movielens/>). The rating are a set of (*user, item, rating, timestamp*) tuples.

```
_____relnCollFilt.py — Latent Feature-based Collaborative Filtering_____
11 import random
12 import matplotlib.pyplot as plt
13 import urllib.request
14 from learnProblem import Learner
15 from utilities import Displayable
16
17 class CF_learner(Learner):
18     def __init__(self,
19                 rating_set,          # a Rating_set object
20                 rating_subset = None, # subset of ratings to be used as training ratings
21                 test_subset = None,   # subset of ratings to be used as test ratings
22                 step_size = 0.01,     # gradient descent step size
23                 reglz = 1.0,          # the weight for the regularization terms
24                 num_features = 10,    # number of hidden features
25                 feature_range = 0.02 # features are initialized to be between
26                                     # -feature_range and feature_range
27             ):
28         self.rating_set = rating_set
29         self.ratings = rating_subset or rating_set.training_ratings # whichever is not empty
30         if test_subset is None:
```

```

31         self.test_ratings = self.rating_set.test_ratings
32     else:
33         self.test_ratings = test_subset
34         self.step_size = step_size
35         self.reglz = reglz
36         self.num_features = num_features
37         self.num_ratings = len(self.ratings)
38         self.ave_rating = (sum(r for (u,i,r,t) in self.ratings)
39                             /self.num_ratings)
40         self.users = {u for (u,i,r,t) in self.ratings}
41         self.items = {i for (u,i,r,t) in self.ratings}
42         self.user_bias = {u:0 for u in self.users}
43         self.item_bias = {i:0 for i in self.items}
44         self.user_feat = {u:[random.uniform(-feature_range,feature_range)
45                               for f in range(num_features)]
46                           for u in self.users}
47         self.item_feat = {i:[random.uniform(-feature_range,feature_range)
48                               for f in range(num_features)]
49                             for i in self.items}
50         self.zeros = [0 for f in range(num_features)]
51         self.iter=0
52
53     def stats(self):
54         self.display(1,"ave sumsq error of mean for training=",
55                     sum((self.ave_rating-rating)**2 for (user,item,rating,timestamp)
56                         in self.ratings)/len(self.ratings))
57         self.display(1,"ave sumsq error of mean for test=",
58                     sum((self.ave_rating-rating)**2 for (user,item,rating,timestamp)
59                         in self.test_ratings)/len(self.test_ratings))
60         self.display(1,"error on training set",
61                     self.evaluate(self.ratings))
62         self.display(1,"error on test set",
63                     self.evaluate(self.test_ratings))

```

*learn* carries out *num\_iter* steps of gradient descent.

relnCollFilt.py — (continued)

```

65     def prediction(self,user,item):
66         """Returns prediction for this user on this item.
67         The use of .get() is to handle users or items not in the training set.
68         """
69         return (self.ave_rating
70                 + self.user_bias.get(user,0) #self.user_bias[user]
71                 + self.item_bias.get(item,0) #self.item_bias[item]
72                 + sum([self.user_feat.get(user,self.zeros)[f]*self.item_feat.get(item,self.zeros)[f]
73                       for f in range(self.num_features)]))
74
75     def learn(self, num_iter = 50):
76         """ do num_iter iterations of gradient descent."""
77         for i in range(num_iter):
78             self.iter += 1

```

```

79         abs_error=0
80         sumsq_error=0
81         for (user,item,rating,timestamp) in random.sample(self.ratings,len(self.ratings)):
82             error = self.prediction(user,item) - rating
83             abs_error += abs(error)
84             sumsq_error += error * error
85             self.user_bias[user] -= self.step_size*error
86             self.item_bias[item] -= self.step_size*error
87             for f in range(self.num_features):
88                 self.user_feat[user][f] -= self.step_size*error*self.item_feat[item][f]
89                 self.item_feat[item][f] -= self.step_size*error*self.user_feat[user][f]
90         for user in self.users:
91             self.user_bias[user] -= self.step_size*self.reglz* self.user_bias[user]
92             for f in range(self.num_features):
93                 self.user_feat[user][f] -= self.step_size*self.reglz*self.user_feat[user][f]
94         for item in self.items:
95             self.item_bias[item] -= self.step_size*self.reglz*self.item_bias[item]
96             for f in range(self.num_features):
97                 self.item_feat[item][f] -= self.step_size*self.reglz*self.item_feat[item][f]
98         self.display(1,"Iteration",self.iter,
99                     "(Ave Abs,AveSumSq) training =",self.evaluate(self.ratings),
100                    "test =",self.evaluate(self.test_ratings))

```

*evaluate* evaluates current predictions on the rating set:

---

```

relnCollFilt.py — (continued)
102     def evaluate(self,ratings):
103         """returns (average_absolute_error, average_sum_squares_error) for ratings
104         """
105         abs_error = 0
106         sumsq_error = 0
107         if not ratings: return (0,0)
108         for (user,item,rating,timestamp) in ratings:
109             error = self.prediction(user,item) - rating
110             abs_error += abs(error)
111             sumsq_error += error * error
112         return abs_error/len(ratings), sumsq_error/len(ratings)

```

### 12.1.1 Alternative Formulation

An alternative formulation is to regularize after each update.

### 12.1.2 Plotting

---

```

relnCollFilt.py — (continued)
114     def plot_predictions(self, examples="test"):
115         """
116         examples is either "test" or "training" or the actual examples
117         """

```

```

118     if examples == "test":
119         examples = self.test_ratings
120     elif examples == "training":
121         examples = self.ratings
122     plt.ion()
123     plt.xlabel("prediction")
124     plt.ylabel("cumulative proportion")
125     self.actuals = [[] for r in range(0,6)]
126     for (user,item,rating,timestamp) in examples:
127         self.actuals[rating].append(self.prediction(user,item))
128     for rating in range(1,6):
129         self.actuals[rating].sort()
130         numrat=len(self.actuals[rating])
131         yvals = [i/numrat for i in range(numrat)]
132         plt.plot(self.actuals[rating], yvals, label="rating="+str(rating))
133     plt.legend()
134     plt.draw()

```

This plots a single feature. Each  $(user, item, rating)$  is plotted where the  $x$ -value is the value of the feature for the user, the  $y$ -value is the value of the feature for the item, and the rating is plotted at this  $(x, y)$  position. That is,  $rating$  is plotted at the  $(x, y)$  position  $(f(user), f(item))$ .

---

```

relnCollFilt.py — (continued)
136     def plot_feature(self,
137         f,                # feature
138         plot_all=False, # true if all points should be plotted
139         num_points=200 # number of random points plotted if not all
140     ):
141         """plot some of the user-movie ratings,
142         if plot_all is true
143         num_points is the number of points selected at random plotted.
144
145         the plot has the users on the x-axis sorted by their value on feature f and
146         with the items on the y-axis sorted by their value on feature f and
147         the ratings plotted at the corresponding x-y position.
148         """
149         plt.ion()
150         plt.xlabel("users")
151         plt.ylabel("items")
152         user_vals = [self.user_feat[u][f]
153             for u in self.users]
154         item_vals = [self.item_feat[i][f]
155             for i in self.items]
156         plt.axis([min(user_vals)-0.02,
157             max(user_vals)+0.05,
158             min(item_vals)-0.02,
159             max(item_vals)+0.05])
160         if plot_all:
161             for (u,i,r,t) in self.ratings:
162                 plt.text(self.user_feat[u][f],

```

```

163         self.item_feat[i][f],
164         str(r))
165     else:
166         for i in range(num_points):
167             (u,i,r,t) = random.choice(self.ratings)
168             plt.text(self.user_feat[u][f],
169                     self.item_feat[i][f],
170                     str(r))
171     plt.show()

```

### 12.1.3 Creating Rating Sets

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)
173 class Rating_set(Displayable):
174     def __init__(self,
175                 date_split=892000000,
176                 local_file=False,
177                 url="http://files.grouplens.org/datasets/movielens/ml-100k/u.data",
178                 file_name="u.data"):
179         self.display(1,"reading...")
180         if local_file:
181             lines = open(file_name,'r')
182         else:
183             lines = (line.decode('utf-8') for line in urllib.request.urlopen(url))
184         all_ratings = (tuple(int(e) for e in line.strip().split('\t'))
185                        for line in lines)
186         self.training_ratings = []
187         self.training_stats = {1:0, 2:0, 3:0, 4:0 ,5:0}
188         self.test_ratings = []
189         self.test_stats = {1:0, 2:0, 3:0, 4:0 ,5:0}
190         for rate in all_ratings:
191             if rate[3] < date_split: # rate[3] is timestamp
192                 self.training_ratings.append(rate)
193                 self.training_stats[rate[2]] += 1
194             else:
195                 self.test_ratings.append(rate)
196                 self.test_stats[rate[2]] += 1
197         self.display(1,"...read:", len(self.training_ratings),"training ratings and",
198                     len(self.test_ratings),"test ratings")
199         tr_users = {user for (user,item,rating,timestamp) in self.training_ratings}
200         test_users = {user for (user,item,rating,timestamp) in self.test_ratings}
201         self.display(1,"users:",len(tr_users),"training,",len(test_users),"test,",
202                     len(tr_users & test_users),"in common")
203         tr_items = {item for (user,item,rating,timestamp) in self.training_ratings}
204         test_items = {item for (user,item,rating,timestamp) in self.test_ratings}
205         self.display(1,"items:",len(tr_items),"training,",len(test_items),"test,",

```

```

206         len(tr_items & test_items),"in common")
207     self.display(1,"Rating statistics for training set: ",self.training_stats)
208     self.display(1,"Rating statistics for test set: ",self.test_stats)

```

Sometimes it is useful to plot a feature 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 features, and may not be useful for other purposes.

```

relnCollFilt.py — (continued)
210 def create_top_subset(self, num_items = 30, num_users = 30):
211     """Returns a subset of the ratings by picking the most rated items,
212     and then the users that have most ratings on these, and then all of the
213     ratings that involve these users and items.
214     """
215     items = {item for (user,item,rating,timestamp) in self.training_ratings}
216
217     item_counts = {i:0 for i in items}
218     for (user,item,rating,timestamp) in self.training_ratings:
219         item_counts[item] += 1
220
221     items_sorted = sorted((item_counts[i],i) for i in items)
222     top_items = items_sorted[-num_items:]
223     set_top_items = set(item for (count, item) in top_items)
224
225     users = {user for (user,item,rating,timestamp) in self.training_ratings}
226     user_counts = {u:0 for u in users}
227     for (user,item,rating,timestamp) in self.training_ratings:
228         if item in set_top_items:
229             user_counts[user] += 1
230
231     users_sorted = sorted((user_counts[u],u)
232                           for u in users)
233     top_users = users_sorted[-num_users:]
234     set_top_users = set(user for (count, user) in top_users)
235     used_ratings = [ (user,item,rating,timestamp)
236                     for (user,item,rating,timestamp) in self.training_ratings
237                     if user in set_top_users and item in set_top_items]
238     return used_ratings
239
240 movielens = Rating_set()
241 learner0 = CF_learner(movielens, num_features = 1)
242 #learner0.learn(50)
243 # learner0.plot_predictions(examples = "training")
244 # learner0.plot_predictions(examples = "test")
245 #learner0.plot_feature(0)
246 #movielens_subset = movielens.create_top_subset(num_items = 20, num_users = 20)
247 #learner1 = CF_learner(movielens, rating_subset=movielens_subset, test_subset=[], num_features=1)
248 #learner1.learn(1000)

```



```
249 | #learner1.plot_feature(0,plot_all=True)
```



# Index

- $\alpha$ - $\beta$  pruning, 183
- A\* search, 39
- action, 79
- agent, 19, 185
- argmax, 15
- assignment, 49, 141
- assumable, 75
- augmented feature, 106
- batched stochastic gradient descent, 123
- blocks world, 81
- Boolean feature, 99
- bottom-up proof, 73
- branch-and-bound search, 43
- class
  - Action\_instance*, 93
  - Agent*, 19
  - Arc*, 32
  - Askable*, 71
  - Assumable*, 75
  - Belief\_network*, 145
  - Boosted\_dataset*, 128
  - Boosting\_learner*, 129
  - Branch\_and\_bound*, 43
  - CF\_learner*, 203
  - CSP*, 50
  - CSP\_from\_STRIPS*, 90
  - Clause*, 71
  - Con\_solver*, 56
  - Constraint*, 49
  - DBN*, 166
  - DBN\_VE\_filter*, 167
  - DBN\_variable*, 165
  - DT\_learner*, 112
  - Data\_from\_file*, 102
  - Data\_set*, 99
  - Data\_set\_augmented*, 106
  - Data\_set\_random*, 110
  - DecisionNetwork*, 170
  - DecisionVariable*, 169
  - Depth\_first\_search*, 46
  - Displayable*, 14
  - EM\_learner*, 134
  - Env\_from\_MDP*, 186
  - Environment*, 20
  - Factor*, 140
  - Factor\_max*, 171
  - Factor\_observed*, 142
  - Factor\_rename*, 145
  - Factor\_stored*, 142

- Factor\_sum*, 143
- Forward\_STRIPS*, 83
- Frontier*, 39
- Gibbs\_sampling*, 157
- Graphical\_model*, 145
- HMM*, 159
- HMM\_VE\_filter*, 160
- HMM\_particle\_filter*, 162
- Healthyenv*, 185
- Inference\_method*, 146
- KB*, 72
- KBA*, 75
- K\_fold\_dataset*, 116
- K\_means\_learner*, 131
- Layer*, 124
- Learner*, 108
- Likelihood\_weighting*, 153
- Linear\_complete\_layer*, 124
- Linear\_learner*, 117
- Linear\_learner\_bsgd*, 123
- MDP*, 174
- Magic\_sum*, 181
- Model\_based\_reinforcement\_learner*, 194
- NN*, 126
- Node*, 179
- POP\_node*, 94
- POP\_search\_from\_STRIPS*, 95
- Particle\_filtering*, 154
- Path*, 34
- Planning\_problem*, 80
- Plot\_env*, 28
- Plot\_prices*, 22
- Prob*, 144
- Q\_learner*, 191
- RL\_agent*, 191
- RL\_env*, 185
- Rating\_set*, 207
- ReLU\_layer*, 125
- Regression\_STRIPS*, 87
- Rejection\_sampling*, 152
- Rob\_body*, 24
- Rob\_env*, 23
- Rob\_middle\_layer*, 26
- Rob\_top\_layer*, 27
- Runtime\_distribution*, 68
- SARSA\_LFA\_learner*, 199
- SLSearcher*, 61
- STRIPS\_domain*, 80
- Sampling\_inference\_method*, 151
- Search\_from\_CSP*, 55
- Search\_problem*, 31
- Search\_problem\_from\_explicit\_graph*, 33
- Search\_with\_AC\_from\_CSP*, 60
- Searcher*, 40
- SearcherMPP*, 42
- Sigmoid\_layer*, 125
- Simple\_game\_env*, 187
- State*, 83
- Strips*, 79
- Subgoal*, 87
- TP\_agent*, 22
- TP\_env*, 20
- Updatable\_priority\_queue*, 66
- Utility*, 169
- VE*, 147
- VE\_DN*, 170
- Variable*, 139
- clause, 71
- collaborative filtering, 203
- condition, 49
- consistency algorithms, 56
- constraint, 49
- constraint satisfaction problem, 49
- copy\_with\_assign, 58
- cross validation, 115
- CSP, 49
  - consistency, 56
  - domain splitting, 58, 59
  - search, 55
  - stochastic local search, 61
- data set, 99
- DBN (dynamic belief network), 165
- decision network, 169
- decision tree learning, 111
- deep learning, 123
- depth-first search, 45
- display, 15

- Displayable, 14
- domain splitting, 58, 59
- dynamic belief network, 165
- EM, 134
- environment, 19, 20, 185
- example, 99
- explicit graph, 32
- factor, 140
- factor\_times, 143
- feature, 99
- file
  - agentEnv.py*, 23
  - agentMiddle.py*, 26
  - agentTop.py*, 27
  - agents.py*, 19
  - cspConsistency.py*, 56
  - cspExamples.py*, 51
  - cspProblem.py*, 49
  - cspSLS.py*, 61
  - cspSearch.py*, 55
  - decnNetworks.py*, 169
  - learnBoosting.py*, 128
  - learnCoordinate.py*, 202
  - learnCrossValidation.py*, 116
  - learnDT.py*, 112
  - learnEM.py*, 134
  - learnKMeans.py*, 131
  - learnLinear.py*, 117
  - learnLinearBSGD.py*, 123
  - learnNN.py*, 124
  - learnNoInputs.py*, 109
  - learnProblem.py*, 99
  - logicAssumables.py*, 75
  - logicBottomUp.py*, 73
  - logicProblem.py*, 71
  - logicTopDown.py*, 75
  - masMiniMax.py*, 182
  - masProblem.py*, 179
  - mdpExamples.py*, 174
  - mdpProblem.py*, 174
  - probDBN.py*, 165
  - probFactors.py*, 140
  - probGraphicalModels.py*, 145
  - probHMM.py*, 159
  - probMCMC.py*, 157
  - probStochSim.py*, 149
  - probVE.py*, 147
  - probVariables.py*, 139
  - pythonDemo.py*, 11
  - relnCollFilt.py*, 203
  - rlFeatures.py*, 199
  - rlModelLearner.py*, 194
  - rlPlot.py*, 189
  - rlProblem.py*, 185
  - rlQLearner.py*, 191
  - rlQTest.py*, 193
  - rlSimpleEnv.py*, 187
  - rlSimpleGameFeatures.py*, 197
  - searchAStar.py*, 39
  - searchBranchAndBound.py*, 43
  - searchDepthFirst.py*, 46
  - searchMPP.py*, 42
  - searchProblem.py*, 31
  - searchTest.py*, 44
  - stripsCSPPlanner.py*, 90
  - stripsForwardPlanner.py*, 83
  - stripsHeuristic.py*, 85
  - stripsPOP.py*, 93
  - stripsProblem.py*, 79
  - stripsRegressionPlanner.py*, 87
  - utilities.py*, 14
- filtering, 160, 162
- forward planner, 83
- game, 179
- Gibbs sampling, 157
- graphical model, 145
- heuristic planning, 85, 89
- hidden Markov model, 159
- hierarchical controller, 23
- HMM
  - exact filtering, 160
  - particle filtering, 162
- HMM (hidden Markov models), 159
- importance sampling, 154
- ipython, 8

- k-means, 130
- knowledge base, 72
- learner, 108
- learning, 99
  - batched stochastic gradient descent, 123
  - cross validation, 115
  - decision tree, 111
  - deep learning, 123
  - EM, 134
  - k-means, 130
  - linear regression, 117
  - linear classification, 117
  - neural network, 123
  - no inputs, 109
  - reinforcement, 185
  - relational, 203
- likelihood weighting, 153
- linear regression, 117
- linear classification, 117
- magic square, 180
- magic-sum game, 180
- Markov Chain Monte Carlo, 157
- Markov decision process, 173
- max\_display\_level, 15
- MCMC, 157
- MDP, 173, 186
- minimax, 179
- minimax algorithm, 182
- minsets, 77
- model-based reinforcement learner, 194
- multiagent system, 179
- multiple path pruning, 42
- naughts and crosses, 180
- neural network, 123
- NotImplementedError, 19
- partial-order planner, 93
- particle filtering, 154
- Particle filtering for HMMs, 162
- planner
  - CSP, 90
  - forward, 83
  - partial order, 93
  - regression, 87
- planning with certainty, 79
- plotting
  - agents in time, 22
  - reinforcement learning, 189
  - robot environment, 28
  - runtime distribution, 68
  - stochastic simulation, 156
- predictor, 100
- probability, 139
- proof
  - bottom-up, 73
  - top-down, 75
- proposition, 71
- Python, 7
- Q learning, 191
- regression planner, 87
- reinforcement learning, 185
  - environment, 185
  - feature-based, 196
  - model-based, 194
  - Q-learning, 191
- rejection sampling, 152
- relational learning, 203
- resampling, 155
- robot
  - body, 24
  - environment, 23
  - middle layer, 26
  - plotting, 28
  - top layer, 27
- robot delivery domain, 80
- runtime, 13
- runtime distribution, 68
- sampling, 149
  - importance sampling, 154
  - belief networks, 151
  - likelihood weighting, 153
  - particle filtering, 154
  - rejection, 152
- scope, 49

- search, 31
  - A\*, 39
  - branch-and-bound, 43
  - depth-first, 45
  - multiple path pruning, 42
- search\_with\_any\_conflict, 63
- search\_with\_var\_pq, 64
- sigmoid, 119
- stochastic local search, 61
  - any-conflict, 63
  - two-stage choice, 64
- stochastic simulation, 149
- test
  - SLS, 69
- tic-tac-toe, 180
- top-down proof, 75
- uncertainty, 139
- unit tests, 16
- updatable priority queue, 66
- value iteration, 175
- variable, 139
- variable elimination (VE), 147
- VE, 147
- yield, 12