Python code for Artificial Intelligence: Foundations of Computational Agents

David L. Poole and Alan K. Mackworth

Version 0.9.0 of June 22, 2021.

http://aipython.org http://artint.info

This document and all the code can be downloaded from http://artint.info/AIPython/ or from http://aipython.org

Contents

Chapter 1

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 istall the latest Python 3 release from `http://python.org/`. This should also install *pip*3. 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 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), with user input in bold:

```
$ ipython -i searchGeneric.py
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.
Testing problem 1:
7 paths have been expanded and 4 paths remain in the frontier
Path found: a --> b --> c --> d --> g
Passed unit test

In [1]: searcher2 = AStarSearcher(searchProblem.acyclic_delivery_problem) #A*

In [2]: searcher2.search()  # find first path
16 paths have been expanded and 5 paths remain in the frontier
Out[2]: o103 --> o109 --> o119 --> o123 --> r123

In [3]: searcher2.search()  # find next path
21 paths have been expanded and 6 paths remain in the frontier
Out[3]: o103 --> b3 --> b4 --> o109 --> o119 --> o123 --> r123

In [4]: searcher2.search()  # find next path
28 paths have been expanded and 5 paths remain in the frontier
Out[4]: o103 --> b3 --> b1 --> b2 --> b4 --> o109 --> o119 --> o123 --> r123

In [5]: searcher2.search()  # find next path
No (more) solutions. Total of 33 paths expanded.
```

```
In [6]:
```

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 Haskell or Lisp, adding a new element to a list, without changing the original list, is a cheap operation. For example if x is a list containing n elements, adding an extra element to the list in Python (using *append*) is fast, but it has the side effect of changing the list x. To construct a new list that contains the elements of x plus a new element, without changing the value of x, entails copying the list, or using a different representation for lists. In the searching code, we will use a different representation for lists for this reason.

1.5 Features of Python

1.5.1 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* for *e* in *iter* 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 (to enumerate integers between ranges) or a file. *cond* is an expression that evaluates to either True or False for each *e*, and *fe* is an expression that will be evaluated for each value of *e* for which *cond* returns *True*.

The result can go in a list or used in another iteration, or can be called directly using *next*. The procedure *next* takes an iterator returns the next element (advancing the iterator) and raises a StopIteration exception if there is no next element. 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. For a local variable in a function, the function uses the last value of the variable when the function is *called*, not the value of the variable when the function was defined (this is called "late binding"). This means if you want to use the value a variable has when the function is created, you need to save the current value of that variable. Whereas Python uses "late binding" by default, the alternative that newcomers often expect is "early binding", where a function uses the value a variable had when the function was defined, can be easily implemented.

Consider the following programs designed to create a list of 5 functions, where the ith function in the list is meant to add i to its argument:[1]

pythonDemo.py — Some tricky examples

```
fun_list1 = []
for i in range(5):
    def fun1(e):
        return e+i
    fun_list1.append(fun1)

fun_list2 = []
for i in range(5):
    def fun2(e,iv=i):
        return e+iv
    fun_list2.append(fun2)

fun_list3 = [lambda e: e+i for i in range(5)]

fun_list4 = [lambda e,iv=i: e+iv for i in range(5)]

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)

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

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

One of the advantages of using the embedded definitions (as in *fun*1 and *fun*2 above) over the lambda is that is it possible to add a `__doc__` string, which is the standard for documenting functions in Python, to the embedded definitions.

1.5.3 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)

```
def myrange(start, stop, step=1):
    """enumerates the values from start in steps of size step that are
    less than stop.
    """
    assert step>0, "only positive steps implemented in myrange"
    i = start
    while i<stop:
        yield i
        i += step

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)

```
def ga(n):
    """generates square of even nonnegative integers less than n"""
    for e in range(n):
        if e%2==0:
            yield e*e
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)

```
def myenumerate(enum):
    for i in range(len(enum)):
        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")
```

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

```
start_time = time.perf_counter(); compute_for_a_while(); end_time = time.perf_counter()
```

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 *timit* 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)

```
import matplotlib.pyplot as plt

def myplot(min,max,step,fun1,fun2):
    plt.ion() # make it interactive
    plt.xlabel("The x axis")
    plt.ylabel("The y axis")
    plt.xscale('linear') # Makes a 'log' or 'linear' scale
    xvalues = range(min,max,step)
    plt.plot(xvalues,[fun1(x) for x in xvalues],
                label="The first fun")
    plt.plot(xvalues,[fun2(x) for x in xvalues], linestyle='--',color='k',
                label=fun2.__doc__) # use the doc string of the function
    plt.legend(loc="upper right") # display the legend

def slin(x):
    """y=2x+7"""
    return 2*x+7
def sqfun(x):
    """y=(x-40)^2/10-20"""
    return (x-40)**2/10-20

# Try the following:
# from pythonDemo import myplot, slin, sqfun
# import matplotlib.pyplot as plt
# myplot(0,100,1,slin,sqfun)
# plt.legend(loc="best")
# import math
# plt.plot([41+40*math.cos(th/10) for th in range(50)],
#          [100+100*math.sin(th/10) for th in range(50)])
# plt.text(40,100,"ellipse?")
# 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

$$self.display(level, to\_print \ldots)$$

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

The definition of *display* is:

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

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

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

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 (nothing that happens repeatedly)

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

3 also display more details

4 and above even more detail

In order to implement more sophisticated visualizations of the algorithm, we add a **visualize** "decorator" to the methods to be visualized. The following code ignores the decorator:

display.py — (continued)

```
def visualize(func):
    """A decorator for algorithms that do interactive visualization.
    Ignored here.
    """
    return func
```

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. *argmaxe* assumes an enumeration; a generator of $(element, value)$ pairs, as for example is generated by the built-in $enumerate(list)$ for lists or $dict.items()$ for dicts.

utilities.py — AIPython useful utilities

```
import random
import math

def argmaxall(gen):
    """gen is a generator of (element,value) pairs, where value is a real.
    argmaxall returns a list of all of the elements with maximal value.
    """
    maxv = -math.inf     # negative infinity
    maxvals = []    # list of maximal elements
    for (e,v) in gen:
        if v>maxv:
            maxvals,maxv = [e], v
        elif v==maxv:
            maxvals.append(e)
    return maxvals

def argmaxe(gen):
    """gen is a generator of (element,value) pairs, where value is a real.
    argmaxe returns an element with maximal value.
    If there are multiple elements with the max value, one is returned at random.
    """
    return random.choice(argmaxall(gen))

def argmax(lst):
    """returns maximum index in a list"""
    return argmaxe(enumerate(lst))
# Try:
# argmax([1,6,3,77,3,55,23])

def argmaxd(dct):
   """returns the arx max of a dictionary dct"""
   return argmaxe(dct.items())
```

```
# Try:
# arxmaxd({2:5,5:9,7:7})
```

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. $\mathit{flip}(p)$ returns True with probability p, and otherwise returns False.

utilities.py — (continued)

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

1.7.4 Dictionary Union

This is now | in Python 3.9, so will be replaced.

The function $\mathit{dict\_union}(d1, d2)$ 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 $\mathit{dict}(d1, **d2)$, but that only works when the keys of $d2$ are strings.

utilities.py — (continued)

```
def dict_union(d1,d2):
    """returns a dictionary that contains the keys of d1 and d2.
    The value for each key that is in d2 is the value from d2,
    otherwise it is the value from d1.
    This does not have side effects.
    """
    d = dict(d1)  # copy d1
    d.update(d2)
    return d
```

1.8 Testing Code

It is important to test code early and test it often. We include a simple form of **unit test**. 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`.

The following code tests `argmax` and `dict_union`, but only when if `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)

```
def test():
    """Test part of utilities"""
    assert argmax(enumerate([1,6,55,3,55,23])) in [2,4]
    assert dict_union({1:4, 2:5, 3:4},{5:7, 2:9}) == {1:4, 2:9, 3:4, 5:7}
    print("Passed unit test in utilities")

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

Chapter 2

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 it 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 overidden in any implemented agent or environment.

agents.py — Agent and Controllers

```
import random

class Agent(object):
    def __init__(self,env):
```

```
        """set up the agent"""
        self.env=env

    def go(self,n):
        """acts for n time steps"""
        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)

```
from display import Displayable
class Environment(Displayable):
    def initial_percepts(self):
        """returns the initial percepts for the agent"""
        raise NotImplementedError("initial_percepts") # abstract method

    def do(self,action):
        """does the action in the environment
        returns the next percept """
        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 $[0, max\_price\_addon)$ plus a linear "inflation". The agent cannot access the price model; it just observes the prices and the amount in stock.

agents.py — (continued)

```
class TP_env(Environment):
```

```
    prices = [234, 234, 234, 234, 255, 255, 275, 275, 211, 211, 211,
    234, 234, 234, 234, 199, 199, 275, 275, 234, 234, 234, 234, 255,
    255, 260, 260, 265, 265, 265, 265, 270, 270, 255, 255, 260, 260,
    265, 265, 150, 150, 265, 265, 270, 270, 255, 255, 260, 260, 265,
    265, 265, 265, 270, 270, 211, 211, 255, 255, 260, 260, 265, 265,
    260, 265, 270, 270, 205, 255, 255, 260, 260, 265, 265, 265, 265,
    270, 270]
    max_price_addon = 20 # maximum of random value added to get price

    def __init__(self):
        """paper buying agent"""
        self.time=0
        self.stock=20
        self.stock_history = [] # memory of the stock history
        self.price_history = [] # memory of the price history

    def initial_percepts(self):
        """return initial percepts"""
        self.stock_history.append(self.stock)
        price = self.prices[0]+random.randrange(self.max_price_addon)
        self.price_history.append(price)
        return {'price': price,
                'instock': self.stock}

    def do(self, action):
        """does action (buy) and returns percepts (price and instock)"""
        used = pick_from_dist({6:0.1, 5:0.1, 4:0.2, 3:0.3, 2:0.2, 1:0.1})
        bought = action['buy']
        self.stock = self.stock+bought-used
        self.stock_history.append(self.stock)
        self.time += 1
        price = (self.prices[self.time%len(self.prices)] # repeating pattern
                 +random.randrange(self.max_price_addon) # plus randomness
                 +self.time//2)                          # plus inflation
        self.price_history.append(price)
        return {'price': price,
                '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)

```
def pick_from_dist(item_prob_dist):
    """ returns a value from a distribution.
    item_prob_dist is an item:probability dictionary, where the
        probabilities sum to 1.
    returns an item chosen in proportion to its probability
    """
    ranreal = random.random()
    for (it,prob) in item_prob_dist.items():
        if ranreal < prob:
```

```
            return it
        else:
            ranreal -= prob
    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)

```
class TP_agent(Agent):
    def __init__(self, env):
        self.env = env
        self.spent = 0
        percepts = env.initial_percepts()
        self.ave = self.last_price = percepts['price']
        self.instock = percepts['instock']

    def go(self, n):
        """go for n time steps
        """
        for i in range(n):
            if self.last_price < 0.9*self.ave and self.instock < 60:
                tobuy = 48
            elif self.instock < 12:
                tobuy = 12
            else:
                tobuy = 0
            self.spent += tobuy*self.last_price
            percepts = env.do({'buy': tobuy})
            self.last_price = percepts['price']
            self.ave = self.ave+(self.last_price-self.ave)*0.05
            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)

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

2.2.3 Plotting

The following plots the price and number in stock history:

agents.py — (continued)

```
import matplotlib.pyplot as plt

class Plot_prices(object):
    """Set up the plot for history of price and number in stock"""
    def __init__(self, ag,env):
        self.ag = ag
        self.env = env
        plt.ion()
        plt.xlabel("Time")
        plt.ylabel("Number in stock.                                        Price.")

    def plot_run(self):
        """plot history of price and instock"""
        num = len(env.stock_history)
        plt.plot(range(num),env.stock_history,label="In stock")
        plt.plot(range(num),env.price_history,label="Price")
        #plt.legend(loc="upper left")
        plt.draw()

# pl = Plot_prices(ag,env)
# 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

```
import math
from agents import Environment

class Rob_env(Environment):
```

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

2.3.2 Body

The body defines everything about the agent body.

agentEnv.py — (continued)

```
import math
from agents import Environment
import matplotlib.pyplot as plt
import time

class Rob_body(Environment):
    def __init__(self, env, init_pos=(0,0,90)):
        """ env is the current environment
        init_pos is a triple of (x-position, y-position, direction)
            direction is in degrees; 0 is to right, 90 is straight-up, etc
        """
        self.env = env
        self.rob_x, self.rob_y, self.rob_dir = init_pos
        self.turning_angle = 18 # degrees that a left makes
        self.whisker_length = 6 # length of the whisker
        self.whisker_angle = 30 # angle of whisker relative to robot
        self.crashed = False
        # The following control how it is plotted
        self.plotting = True   # whether the trace is being plotted
        self.sleep_time = 0.05  # time between actions (for real-time plotting)
        # The following are data structures maintained:
        self.history = [(self.rob_x, self.rob_y)] # history of (x,y) positions
        self.wall_history = []  # history of hitting the wall

    def percepts(self):
        return {'rob_x_pos':self.rob_x, 'rob_y_pos':self.rob_y,
                'rob_dir':self.rob_dir, 'whisker':self.whisker() , 'crashed':self.crashed}
    initial_percepts = percepts # use percept function for initial percepts too

    def do(self,action):
        """ action is {'steer':direction}
        direction is 'left', 'right' or 'straight'
        """
        if self.crashed:
            return self.percepts()
        direction = action['steer']
        compass_deriv = {'left':1,'straight':0,'right':-1}[direction]*self.turning_angle
        self.rob_dir = (self.rob_dir + compass_deriv +360)%360 # make in range [0,360)
        rob_x_new = self.rob_x + math.cos(self.rob_dir*math.pi/180)
```

```
        rob_y_new = self.rob_y + math.sin(self.rob_dir*math.pi/180)
        path = ((self.rob_x,self.rob_y),(rob_x_new,rob_y_new))
        if any(line_segments_intersect(path,wall) for wall in self.env.walls):
            self.crashed = True
            if self.plotting:
                plt.plot([self.rob_x],[self.rob_y],"r*",markersize=20.0)
                plt.draw()
        self.rob_x, self.rob_y = rob_x_new, rob_y_new
        self.history.append((self.rob_x, self.rob_y))
        if self.plotting and not self.crashed:
            plt.plot([self.rob_x],[self.rob_y],"go")
            plt.draw()
            plt.pause(self.sleep_time)
        return self.percepts()
```

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

agentEnv.py — (continued)

```
    def whisker(self):
        """returns true whenever the whisker sensor intersects with a wall
        """
        whisk_ang_world = (self.rob_dir-self.whisker_angle)*math.pi/180
            # angle in radians in world coordinates
        wx = self.rob_x + self.whisker_length * math.cos(whisk_ang_world)
        wy = self.rob_y + self.whisker_length * math.sin(whisk_ang_world)
        whisker_line = ((self.rob_x,self.rob_y),(wx,wy))
        hit = any(line_segments_intersect(whisker_line,wall)
                    for wall in self.env.walls)
        if hit:
            self.wall_history.append((self.rob_x, self.rob_y))
            if self.plotting:
                plt.plot([self.rob_x],[self.rob_y],"ro")
                plt.draw()
        return hit

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

```
    ca = (db*(y0b-y0a)-eb*(x0b-x0a))/denom # position along line a
    return 0<=ca<=1

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

2.3.3 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*$(\cdot)$. It also is told the position to go to and the timeout. Thus it also has to implement *do*$(\cdot)$.

agentMiddle.py — Middle Layer

```
from agents import Environment
import math

class Rob_middle_layer(Environment):
    def __init__(self,env):
        self.env=env
        self.percepts = env.initial_percepts()
        self.straight_angle = 11 # angle that is close enough to straight ahead
        self.close_threshold = 2 # distance that is close enough to arrived
        self.close_threshold_squared = self.close_threshold**2 # just compute it once

    def initial_percepts(self):
        return {}

    def do(self, action):
        """action is {'go_to':target_pos,'timeout':timeout}
        target_pos is (x,y) pair
        timeout is the number of steps to try
        returns {'arrived':True} when arrived is true
             or {'arrived':False} if it reached the timeout
        """
        if 'timeout' in action:
            remaining = action['timeout']
        else:
            remaining = -1  # will never reach 0
        target_pos = action['go_to']
        arrived = self.close_enough(target_pos)
        while not arrived and remaining != 0:
            self.percepts = self.env.do({"steer":self.steer(target_pos)})
            remaining -= 1
            arrived = self.close_enough(target_pos)
        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)

```
    def steer(self,target_pos):
        if self.percepts['whisker']:
            self.display(3,'whisker on', self.percepts)
            return "left"
        else:
            gx,gy = target_pos
            rx,ry = self.percepts['rob_x_pos'],self.percepts['rob_y_pos']
            goal_dir = math.acos((gx-rx)/math.sqrt((gx-rx)*(gx-rx)
                                           +(gy-ry)*(gy-ry)))*180/math.pi
            if ry>gy:
                goal_dir = -goal_dir
            goal_from_rob = (goal_dir - self.percepts['rob_dir']+540)%360-180
            assert -180 < goal_from_rob <= 180
            if goal_from_rob > self.straight_angle:
                return "left"
            elif goal_from_rob < -self.straight_angle:
                return "right"
            else:
                return "straight"

    def close_enough(self,target_pos):
        gx,gy = target_pos
        rx,ry = self.percepts['rob_x_pos'],self.percepts['rob_y_pos']
        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

```
from agentMiddle import Rob_middle_layer
from agents import Environment

class Rob_top_layer(Environment):
    def __init__(self, middle, timeout=200, locations = {'mail':(-5,10),
                          'o103':(50,10), 'o109':(100,10),'storage':(101,51)} ):
        """middle is the middle layer
        timeout is the number of steps the middle layer goes before giving up
        locations is a loc:pos dictionary
            where loc is a named location, and pos is an (x,y) position.
        """
        self.middle = middle
        self.timeout = timeout # number of steps before the middle layer should give up
        self.locations = locations

```

```
    def do(self,plan):
        """carry out actions.
        actions is of the form {'visit':list_of_locations}
        It visits the locations in turn.
        """
        to_do = plan['visit']
        for loc in to_do:
            position = self.locations[loc]
            arrived = self.middle.do({'go_to':position, 'timeout':self.timeout})
            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)

```
import matplotlib.pyplot as plt

class Plot_env(object):
    def __init__(self, body,top):
        """sets up the plot
        """
        self.body = body
        plt.ion()
        plt.clf()
        plt.axes().set_aspect('equal')
        for wall in body.env.walls:
            ((x0,y0),(x1,y1)) = wall
            plt.plot([x0,x1],[y0,y1],"-k",linewidth=3)
        for loc in top.locations:
            (x,y) = top.locations[loc]
            plt.plot([x],[y],"k<")
            plt.text(x+1.0,y+0.5,loc) # print the label above and to the right
        plt.plot([body.rob_x],[body.rob_y],"go")
        plt.draw()

    def plot_run(self):
        """plots the history after the agent has finished.
        This is typically only used if body.plotting==False
        """
        xs,ys = zip(*self.body.history)
        plt.plot(xs,ys,"go")
        wxs,wys = zip(*self.body.wall_history)
        plt.plot(wxs,wys,"ro")
        #plt.draw()
```

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

agentTop.py — (continued)

```
from agentEnv import Rob_body, Rob_env

env = Rob_env({((20,0),(30,20)), ((70,-5),(70,25))})
body = Rob_body(env)
middle = Rob_middle_layer(body)
top = Rob_top_layer(middle)

# try:
# pl=Plot_env(body,top)
# top.do({'visit':['o109','storage','o109','o103']})
# You can directly control the middle layer:
# middle.do({'go_to':(30,-10), 'timeout':200})
# 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)

```
# Robot Trap for which the current controller cannot escape:
trap_env = Rob_env({((10,-21),(10,0)), ((10,10),(10,31)), ((30,-10),(30,0)),
                  ((30,10),(30,20)), ((50,-21),(50,31)), ((10,-21),(50,-21)),
                  ((10,0),(30,0)), ((10,10),(30,10)), ((10,31),(50,31))})
trap_body = Rob_body(trap_env,init_pos=(-1,0,90))
trap_middle = Rob_middle_layer(trap_body)
trap_top = Rob_top_layer(trap_middle,locations={'goal':(71,0)})

# Robot trap exercise:
# pl=Plot_env(trap_body,trap_top)
# trap_top.do({'visit':['goal']})
```

Chapter 3

Searching for Solutions

3.1 Representing Search Problems

A search problem consists of:

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

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

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

searchProblem.py — representations of search problems

```
class Search_problem(object):
    """A search problem consists of:
    * a start node
    * a neighbors function that gives the neighbors of a node
    * a specification of a goal
    * a (optional) heuristic function.
```

```
    The methods must be overridden to define a search problem."""

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

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

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

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

The neighbors is a list 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)

```
class Arc(object):
    """An arc has a from_node and a to_node node and a (non-negative) cost"""
    def __init__(self, from_node, to_node, cost=1, action=None):
        assert cost >= 0, ("Cost cannot be negative for"+
                           str(from_node)+"->"+str(to_node)+", cost: "+str(cost))
        self.from_node = from_node
        self.to_node = to_node
        self.action = action
        self.cost=cost

    def __repr__(self):
        """string representation of an arc"""
        if self.action:
            return str(self.from_node)+" --"+str(self.action)+"--> "+str(self.to_node)
        else:
            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 start node
- 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 node, the goal predicate, the neighbors function and the heuristic function.

searchProblem.py — (continued)

```
class Search_problem_from_explicit_graph(Search_problem):
    """A search problem consists of:
    * a list or set of nodes
    * a list or set of arcs
    * a start node
    * a list or set of goal nodes
    * a dictionary that maps each node into its heuristic value.
    * a dictionary that maps each node into its (x,y) position
    """

    def __init__(self, nodes, arcs, start=None, goals=set(), hmap={}, positions={}):
        self.neighs = {}
        self.nodes = nodes
        for node in nodes:
            self.neighs[node]=[]
        self.arcs = arcs
        for arc in arcs:
            self.neighs[arc.from_node].append(arc)
        self.start = start
        self.goals = goals
        self.hmap = hmap
        self.positions = positions

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

    def is_goal(self,node):
        """is True if node is a goal"""
        return node in self.goals

    def neighbors(self,node):
        """returns the neighbors of node"""
        return self.neighs[node]

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

```
        else:
            return 0

    def __repr__(self):
        """returns a string representation of the search problem"""
        res=""
        for arc in self.arcs:
            res += str(arc)+". "
        return res
```

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

searchProblem.py — (continued)

```
    def neighbor_nodes(self,node):
        """returns an iterator over the neighbors of node"""
        return (path.to_node for path in self.neighs[node])
```

3.1.2 Paths

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

A path is either:

- a node (representing a path of length 0) or
- 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)

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

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

    def end(self):
```

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

    def nodes(self):
        """enumerates the nodes for the path.
        This starts at the end and enumerates nodes in the path backwards."""
        current = self
        while current.arc is not None:
            yield current.arc.to_node
            current = current.initial
        yield current.initial

    def initial_nodes(self):
        """enumerates the nodes for the path before the end node.
        This starts at the end and enumerates nodes in the path backwards."""
        if self.arc is not None:
            for nd in self.initial.nodes(): yield nd # could be "yield from"

    def __repr__(self):
        """returns a string representation of a path"""
        if self.arc is None:
            return str(self.initial)
        elif self.arc.action:
            return (str(self.initial)+"\n --"+str(self.arc.action)
                    +"--> "+str(self.arc.to_node))
        else:
            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 `searchGeneric`) will need to be changed if this is changed.

searchProblem.py — (continued)

```
problem1 = Search_problem_from_explicit_graph(
    {'a','b','c','d','g'},
    [Arc('a','c',1), Arc('a','b',3), Arc('c','d',3), Arc('c','b',1),
        Arc('b','d',1), Arc('b','g',3), Arc('d','g',1)],
    start = 'a',
    goals = {'g'},
    positions={'a': (0, 0), 'b': (1, 1), 'c': (0,1), 'd': (1,2), 'g': (2,2)})
```

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)

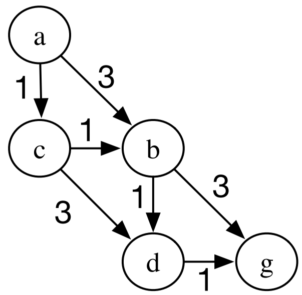


Figure 3.1: problem1

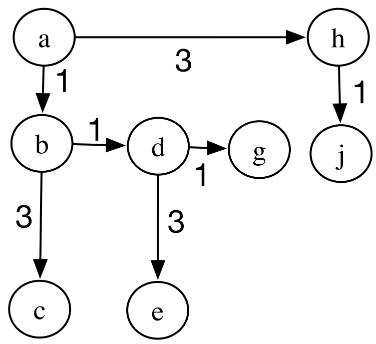


Figure 3.2: problem2

```
problem2 = Search_problem_from_explicit_graph(
    {'a','b','c','d','e','g','h','j'},
    [Arc('a','b',1), Arc('b','c',3), Arc('b','d',1), Arc('d','e',3),
        Arc('d','g',1), Arc('a','h',3), Arc('h','j',1)],
    start = 'a',
    goals = {'g'},
    positions={'a': (0, 0), 'b': (0, 1), 'c': (0,4), 'd': (1,1), 'e': (1,4),
                   'g': (2,1), 'h': (3,0), 'j': (3,1)})
```

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

searchProblem.py — (continued)

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

```
acyclic_delivery_problem = Search_problem_from_explicit_graph(
    {'mail','ts','o103','o109','o111','b1','b2','b3','b4','c1','c2','c3',
     'o125','o123','o119','r123','storage'},
     [Arc('ts','mail',6),
        Arc('o103','ts',8),
        Arc('o103','b3',4),
        Arc('o103','o109',12),
        Arc('o109','o119',16),
        Arc('o109','o111',4),
        Arc('b1','c2',3),
        Arc('b1','b2',6),
        Arc('b2','b4',3),
        Arc('b3','b1',4),
        Arc('b3','b4',7),
        Arc('b4','o109',7),
        Arc('c1','c3',8),
        Arc('c2','c3',6),
        Arc('c2','c1',4),
        Arc('o123','o125',4),
        Arc('o123','r123',4),
        Arc('o119','o123',9),
        Arc('o119','storage',7)],
    start = 'o103',
    goals = {'r123'},
    hmap = {
        'mail' : 26,
        'ts' : 23,
        'o103' : 21,
        'o109' : 24,
        'o111' : 27,
        'o119' : 11,
        'o123' : 4,
        'o125' : 6,
        'r123' : 0,
        'b1' : 13,
        'b2' : 15,
        'b3' : 17,
        'b4' : 18,
        'c1' : 6,
        'c2' : 10,
        'c3' : 12,
        'storage' : 12
        }
    )
```

The `cyclic_delivery_problem` is the delivery problem described in Example 3.8 and shown in Figure 3.6 of the textbook. This is the same as `acyclic_delivery_problem`, but almost every arc also has its inverse.

searchProblem.py — (continued)

```
cyclic_delivery_problem = Search_problem_from_explicit_graph(
    {'mail','ts','o103','o109','o111','b1','b2','b3','b4','c1','c2','c3',
     'o125','o123','o119','r123','storage'},
     [  Arc('ts','mail',6), Arc('mail','ts',6),
        Arc('o103','ts',8), Arc('ts','o103',8),
        Arc('o103','b3',4),
        Arc('o103','o109',12), Arc('o109','o103',12),
        Arc('o109','o119',16), Arc('o119','o109',16),
        Arc('o109','o111',4), Arc('o111','o109',4),
        Arc('b1','c2',3),
        Arc('b1','b2',6), Arc('b2','b1',6),
        Arc('b2','b4',3), Arc('b4','b2',3),
        Arc('b3','b1',4), Arc('b1','b3',4),
        Arc('b3','b4',7), Arc('b4','b3',7),
        Arc('b4','o109',7),
        Arc('c1','c3',8), Arc('c3','c1',8),
        Arc('c2','c3',6), Arc('c3','c2',6),
        Arc('c2','c1',4), Arc('c1','c2',4),
        Arc('o123','o125',4), Arc('o125','o123',4),
        Arc('o123','r123',4), Arc('r123','o123',4),
        Arc('o119','o123',9), Arc('o123','o119',9),
        Arc('o119','storage',7), Arc('storage','o119',7)],
    start = 'o103',
    goals = {'r123'},
    hmap = {
        'mail' : 26,
        'ts' : 23,
        'o103' : 21,
        'o109' : 24,
        'o111' : 27,
        'o119' : 11,
        'o123' : 4,
        'o125' : 6,
        'r123' : 0,
        'b1' : 13,
        'b2' : 15,
        'b3' : 17,
        'b4' : 18,
        'c1' : 6,
        'c2' : 10,
        'c3' : 12,
        'storage' : 12
        }
    )
```

3.2 Generic Searcher and Variants

To run the search demos, in folder "aipython", load "searchGeneric.py" , using e.g., `ipython -i searchGeneric.py`, and copy and paste the example queries at the bottom of that file. This requires Python 3.

3.2.1 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.

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

```
from display import Displayable, visualize

class Searcher(Displayable):
    """returns a searcher for a problem.
    Paths can be found by repeatedly calling search().
    This does depth-first search unless overridden
    """
    def __init__(self, problem):
        """creates a searcher from a problem
        """
        self.problem = problem
        self.initialize_frontier()
        self.num_expanded = 0
        self.add_to_frontier(Path(problem.start_node()))
        super().__init__()

    def initialize_frontier(self):
        self.frontier = []

    def empty_frontier(self):
        return self.frontier == []

    def add_to_frontier(self,path):
        self.frontier.append(path)

    @visualize
    def search(self):
        """returns (next) path from the problem's start node
        to a goal node.
        Returns None if no path exists.
        """
        while not self.empty_frontier():
            path = self.frontier.pop()
            self.display(2, "Expanding:",path,"(cost:",path.cost,")")
            self.num_expanded += 1
```

```
            if self.problem.is_goal(path.end()): # solution found
                self.display(1, self.num_expanded, "paths have been expanded and",
                             len(self.frontier), "paths remain in the frontier")
                self.solution = path # store the solution found
                return path
            else:
                neighs = self.problem.neighbors(path.end())
                self.display(3,"Neighbors are", neighs)
                for arc in reversed(list(neighs)):
                    self.add_to_frontier(Path(path,arc))
                self.display(3,"Frontier:",self.frontier)
        self.display(1,"No (more) solutions. Total of",
                     self.num_expanded,"paths expanded.")
```

Note that this reverses the neigbours so that it implements depth-first search in an intutive manner (expanding the first neighbor first), and *list* is needed if the neighboure are generated. Reversing the neighbours might not be required for other methods. The calls to *reversed* and *list* can be removed, and the algothihm still implements depth-fist search.

Exercise 3.1 When it returns a path, the algorithm can be used to find another path by calling *search*() again. However, it does not find other paths that go through one goal node to another. Explain why, and change the code so that it can find such paths when *search*() is called again.

3.2.2 Frontier as a Priority Queue

In many of the search algorithms, such as A^* and other best-first searchers, the frontier is implemented as a priority queue. 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.

searchGeneric.py — (continued)

```
import heapq      # part of the Python standard library
from searchProblem import Path

class FrontierPQ(object):
```

```
    """A frontier consists of a priority queue (heap), frontierpq, of
        (value, index, path) triples, where
    * value is the value we want to minimize (e.g., path cost + h).
    * index is a unique index for each element
    * path is the path on the queue
    Note that the priority queue always returns the smallest element.
    """

    def __init__(self):
        """constructs the frontier, initially an empty priority queue
        """
        self.frontier_index = 0 # the number of items ever added to the frontier
        self.frontierpq = [] # the frontier priority queue

    def empty(self):
        """is True if the priority queue is empty"""
        return self.frontierpq == []

    def add(self, path, value):
        """add a path to the priority queue
        value is the value to be minimized"""
        self.frontier_index += 1 # get a new unique index
        heapq.heappush(self.frontierpq,(value, -self.frontier_index, path))

    def pop(self):
        """returns and removes the path of the frontier with minimum value.
        """
        (_,_,path) = heapq.heappop(self.frontierpq)
        return path
```

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

searchGeneric.py — (continued)

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

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

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

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

3.2.3 A^* Search

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

searchGeneric.py — (continued)

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

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

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

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

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

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

searchGeneric.py — (continued)

```
import searchProblem as searchProblem

def test(SearchClass, problem=searchProblem.problem1, solutions=[['g','d','b','c','a']] ):
    """Unit test for aipython searching algorithms.
    SearchClass is a class that takes a problemm and implements search()
    problem is a search problem
    solutions is a list of optimal solutions
    """
    print("Testing problem 1:")
    schr1 = SearchClass(problem)
    path1 = schr1.search()
    print("Path found:",path1)
    assert path1 is not None, "No path is found in problem1"
    assert list(path1.nodes()) in solutions, "Shortest path not found in problem1"
    print("Passed unit test")

if __name__ == "__main__":
    #test(Searcher)
    test(AStarSearcher)

# example queries:
# searcher1 = Searcher(searchProblem.acyclic_delivery_problem) # DFS
# searcher1.search() # find first path
```

```
# searcher1.search() # find next path
# searcher2 = AStarSearcher(searchProblem.acyclic_delivery_problem) # A*
# searcher2.search() # find first path
# searcher2.search() # find next path
# searcher3 = Searcher(searchProblem.cyclic_delivery_problem) # DFS
# searcher3.search() # find first path with DFS. What do you expect to happen?
# searcher4 = AStarSearcher(searchProblem.cyclic_delivery_problem) # A*
# searcher4.search() # find first path
```

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

Exercise 3.3 In the *add* method in *FrontierPQ* what does the "-" in front of *frontier_index* do? When there are multiple paths with the same f-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?

Exercise 3.4 The searcher acts like a Python iterator, in that it returns one value (here a path) and then returns other values (paths) on demand, but does not implement the iterator interface. Change the code so it implements the iterator interface. What does this enable us to do?

3.2.4 Multiple Path Pruning

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

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

searchMPP.py — Searcher with multiple-path pruning

```
from searchGeneric import AStarSearcher, visualize
from searchProblem import Path

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

    @visualize
    def search(self):
        """returns next path from an element of problem's start nodes
        to a goal node.
        Returns None if no path exists.
```

```
        """
        while not self.empty_frontier():
            path = self.frontier.pop()
            if path.end() not in self.explored:
                self.display(2, "Expanding:",path,"(cost:",path.cost,")")
                self.explored.add(path.end())
                self.num_expanded += 1
                if self.problem.is_goal(path.end()):
                    self.display(1, self.num_expanded, "paths have been expanded and",
                            len(self.frontier), "paths remain in the frontier")
                    self.solution = path # store the solution found
                    return path
                else:
                    neighs = self.problem.neighbors(path.end())
                    self.display(3,"Neighbors are", neighs)
                    for arc in neighs:
                        self.add_to_frontier(Path(path,arc))
                    self.display(3,"Frontier:",self.frontier)
        self.display(1,"No (more) solutions. Total of",
                     self.num_expanded,"paths expanded.")

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

import searchProblem
# searcherMPPcdp = SearcherMPP(searchProblem.cyclic_delivery_problem)
# print(searcherMPPcdp.search()) # find first path
```

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

3.3 Branch-and-bound Search

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

Depth-first search methods do not need 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

```
from searchProblem import Path
from searchGeneric import Searcher
from display import Displayable, visualize

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

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

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

Here is a unit test and some queries:

searchBranchAndBound.py — (continued)

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

# Example queries:
import searchProblem
# searcherb1 = DF_branch_and_bound(searchProblem.acyclic_delivery_problem)
# print(searcherb1.search())    # find optimal path
# searcherb2 = DF_branch_and_bound(searchProblem.cyclic_delivery_problem, bound=100)
# print(searcherb2.search())    # find optimal path
```

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.

Exercise 3.7 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

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

DF_branch_and_bound.max_display_level = 1
Searcher.max_display_level = 1

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

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

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

```

```
    bound = asearcher.solution.cost+0.01
    print("\nBranch and bound (with too-good initial bound of", bound,")")
    tbb = DF_branch_and_bound(problem,bound) # cheating!!!!
    print("Path found:",tbb.search()," cost=",tbb.solution.cost)
    print("Rerunning B&B")
    print("Path found:",tbb.search())

    bbound = asearcher.solution.cost*2+10
    print("\nBranch and bound (with not-very-good initial bound of", bbound, ")")
    tbb2 = DF_branch_and_bound(problem,bbound) # cheating!!!!
    print("Path found:",tbb2.search()," cost=",tbb2.solution.cost)
    print("Rerunning B&B")
    print("Path found:",tbb2.search())

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


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

Chapter 4

Reasoning with Constraints

4.1 Constraint Satisfaction Problems

4.1.1 Constraints

A **variable** is a string or any value that is printable and can be the key of a Python dictionary.

A **constraint** consists of a list (or tuple) of variables and a condition.

- The tuple (or list) of variables is called the **scope**.
- The **condition** is a Boolean function that takes the same number of arguments as there are variables in the scope. The condition must have 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

```python
class Constraint(object):
    """A Constraint consists of
    * scope: a tuple of variables
    * condition: a function that can applied to a tuple of values
    * string: a string for printing the constraints. All of the strings must be unique.
    for the variables
    """
    def __init__(self, scope, condition, string=None):
        self.scope = scope
        self.condition = condition
        if string is None:
            self.string = self.condition.__name__ + str(self.scope)
        else:
```

```
        self.string = string

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

An **assignment** is a *variable:value* dictionary.

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

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

cspProblem.py — (continued)

```
    def holds(self,assignment):
        """returns the value of Constraint con evaluated in assignment.

        precondition: all variables are assigned in assignment
        """
        return self.condition(*tuple(assignment[v] for v in self.scope))
```

4.1.2 CSPs

A constraint satisfaction problem (CSP) requires:

- *domains*: a dictionary that maps variables to the set of possible values. Thus *domains*[*var*] is the domain of variable *var*.
- *constaraints*: a set or list of constraints.

Other properties are inferred from these:

- *variables* is the set of variables. The variables can be enumerated by using "for var in domains" because iterating over a dictionary gives the keys, which in this case are the variables.
- *var_to_const* is a mapping from variables to set of constraints, such that *var_to_const*[*var*] is the set of constraints with *var* in the scope.

cspProblem.py — (continued)

```
class CSP(object):
    """A CSP consists of
    * domains, a dictionary that maps each variable to its domain
    * constraints, a list of constraints
    * variables, a set of variables
```

```
    * var_to_const, a variable to set of constraints dictionary
    """
    def __init__(self, domains, constraints, positions={}):
        """domains is a variable:domain dictionary
        constraints is a list of constriants
        """
        self.variables = set(domains)
        self.domains = domains
        self.constraints = constraints
        self.positions = positions
        self.var_to_const = {var:set() for var in self.variables}
        for con in constraints:
            for var in con.scope:
                self.var_to_const[var].add(con)

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

    def __repr__(self):
        """more detailed string representation of CSP"""
        return "CSP("+str(self.domains)+", "+str([str(c) for c in self.constraints])+")"
```

csp.consistent(*assignment*) returns true if the assignment is consistent with each of the constraints in *csp* (i.e., all of the constraints that can be evaluated evaluate to true). Note that this is a local consistency with each constraint; it does *not* imply the CSP is consistent or has a solution.

cspProblem.py — (continued)

```
    def consistent(self,assignment):
        """assignment is a variable:value dictionary
        returns True if all of the constraints that can be evaluated
                        evaluate to True given assignment.
        """
        return all(con.holds(assignment)
                    for con in self.constraints
                    if 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. For example, if $f = ne\_(3)$, then $f(2)$ is True and $f(3)$ is False. That is, $ne\_(x)(y)$ is true when $x \neq y$. Allowing a function of multiple arguments to use its arguments one at a time is called **currying**, after the logician Haskell Curry. Functions used as conditions in constraints require names (so they can be printed).

cspExamples.py — Example CSPs

```
from cspProblem import CSP, Constraint
```

```python
from operator import lt,ne,eq,gt

def ne_(val):
    """not equal value"""
    # nev = lambda x: x != val # alternative definition
    # nev = partial(neq,val)  # another alternative definition
    def nev(x):
        return val != x
    nev.__name__ = str(val)+"!="   # name of the function
    return nev
```

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

cspExamples.py — (continued)

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

The CSP, *csp*0 has variables X, Y and Z, each with domain $\{1, 2, 3\}$. The constraints are $X < Y$ and $Y < Z$.

cspExamples.py — (continued)

```python
csp0 = CSP({'X':{1,2,3},'Y':{1,2,3}, 'Z':{1,2,3}},
          [ Constraint(['X','Y'],lt),
            Constraint(['Y','Z'],lt)])
```

The CSP, *csp*1 has variables A, B and C, each with domain $\{1, 2, 3, 4\}$. The constraints are $A < B$, $B \neq 2$ and $B < C$. This is slightly more interesting than *csp*0 as it has more solutions. This example is used in the unit tests, and so if it is changed, the unit tests need to be changed.

cspExamples.py — (continued)

```python
C0 = Constraint(['A','B'], lt, "A < B")
C1 = Constraint(['B'], ne_(2), "B != 2")
C2 = Constraint(['B','C'], lt, "B < C")
csp1 = CSP({'A':{1,2,3,4},'B':{1,2,3,4}, 'C':{1,2,3,4}},
          [C0, C1, C2],
          positions={"A": (1, 0),
                     "B": (3, 0),
                     "C": (5, 0),
                     "A < B": (2, 1),
                     "B < C": (4, 1),
                     "B != 2": (3, 2)})
```

The next CSP, *csp*2 is Example 4.9 of the textbook; the domain consistent network (after applying the unary constriants) is shown in Figure 4.1.

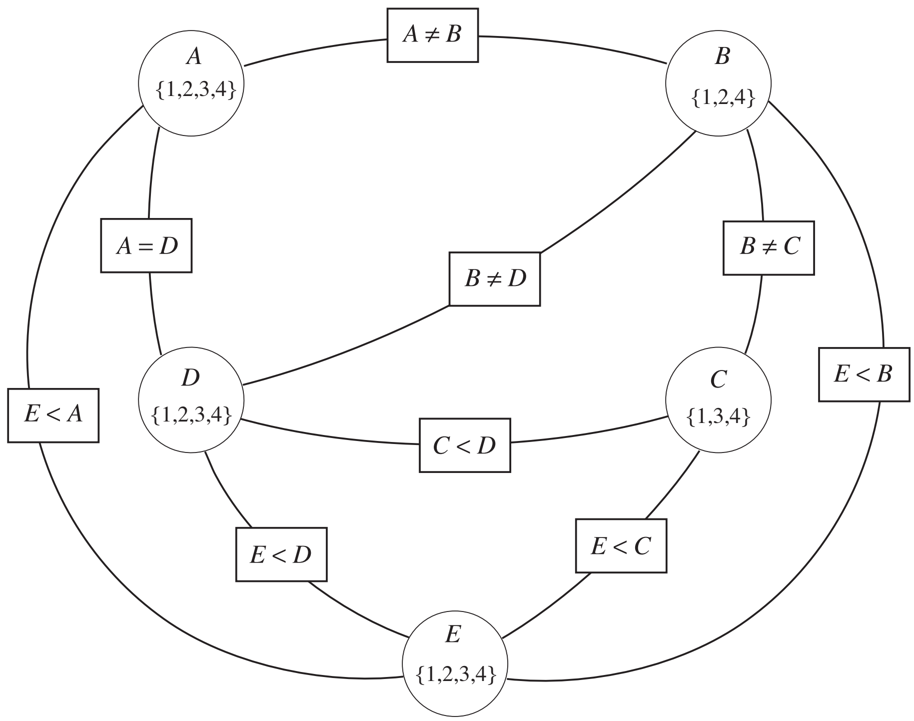


Figure 4.1: Domain-consistent constraint network (*csp*2).

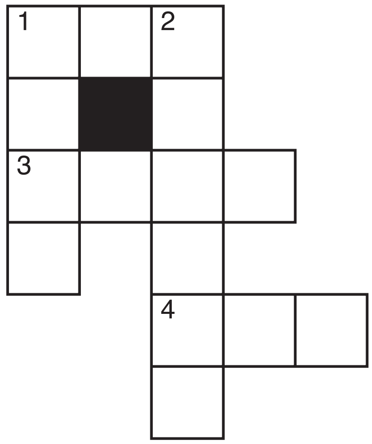
cspExamples.py — (continued)

```
csp2 = CSP({'A':{1,2,3,4},'B':{1,2,3,4}, 'C':{1,2,3,4},
           'D':{1,2,3,4}, 'E':{1,2,3,4}},
          [ Constraint(['B'], ne_(3), "B != 3"),
           Constraint(['C'], ne_(2), "C != 2"),
           Constraint(['A','B'], ne, "A != B"),
           Constraint(['B','C'], ne, "A != C"),
           Constraint(['C','D'], lt, "C < D"),
           Constraint(['A','D'], eq, "A = D"),
           Constraint(['A','E'], gt, "A > E"),
           Constraint(['B','E'], gt, "B > E"),
           Constraint(['C','E'], gt, "C > E"),
           Constraint(['D','E'], gt, "D > E"),
           Constraint(['B','D'], ne, "B != D")])
```

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

cspExamples.py — (continued)

```
csp3 = CSP({'A':{1,2,3,4},'B':{1,2,3,4}, 'C':{1,2,3,4},
           'D':{1,2,3,4}, 'E':{1,2,3,4}},
          [Constraint(['A','B'], ne, "A != B"),
           Constraint(['A','D'], lt, "A < D"),
```



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

Figure 4.2: A crossword puzzle to be solved

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

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

cspExamples.py — (continued)

```
def adjacent(x,y):
   """True when x and y are adjacent numbers"""
   return abs(x-y) == 1

csp4 = CSP({'A':{1,2,3,4,5},'B':{1,2,3,4,5}, 'C':{1,2,3,4,5},
            'D':{1,2,3,4,5}, 'E':{1,2,3,4,5}},
           [Constraint(['A','B'], adjacent, "adjacent(A,B)"),
            Constraint(['B','C'], adjacent, "adjacent(B,C)"),
            Constraint(['C','D'], adjacent, "adjacent(C,D)"),
            Constraint(['D','E'], adjacent, "adjacent(D,E)"),
            Constraint(['A','C'], ne, "A != C"),
            Constraint(['B','D'], ne, "A != D"),
            Constraint(['C','E'], ne, "C != E")])
```

The following examples represent the crossword shown in Figure 4.2.

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

cspExamples.py — (continued)

```
def meet_at(p1,p2):
    """returns a function of two words that is true when the words intersect at postions p1, p2.
    The positions are relative to the words; starting at position 0.
    meet_at(p1,p2)(w1,w2) is true if the same letter is at position p1 of word w1
         and at position p2 of word w2.
    """
    def meets(w1,w2):
        return w1[p1] == w2[p2]
    meets.__name__ = "meet_at("+str(p1)+','+str(p2)+')'
    return meets

crossword1 = CSP({'one_across':{'ant', 'big', 'bus', 'car', 'has'},
                  'one_down':{'book', 'buys', 'hold', 'lane', 'year'},
                  'two_down':{'ginger', 'search', 'symbol', 'syntax'},
                  'three_across':{'book', 'buys', 'hold', 'land', 'year'},
                  'four_across':{'ant', 'big', 'bus', 'car', 'has'}},
                  [Constraint(['one_across','one_down'], meet_at(0,0)),
                   Constraint(['one_across','two_down'], meet_at(2,0)),
                   Constraint(['three_across','two_down'], meet_at(2,2)),
                   Constraint(['three_across','one_down'], meet_at(0,2)),
                   Constraint(['four_across','two_down'], meet_at(0,4))])
```

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

cspExamples.py — (continued)

```
words = {'ant', 'big', 'bus', 'car', 'has','book', 'buys', 'hold',
         'lane', 'year', 'ginger', 'search', 'symbol', 'syntax'}

def is_word(*letters, words=words):
    """is true if the letters concatenated form a word in words"""
    return "".join(letters) in words

letters = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
  "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y",
  "z"]

crossword1d = CSP({'p00':letters, 'p10':letters, 'p20':letters, # first row
                   'p01':letters, 'p21':letters, # second row
                   'p02':letters, 'p12':letters, 'p22':letters, 'p32':letters, # third row
                   'p03':letters, 'p23':letters, #fourth row
                   'p24':letters, 'p34':letters, 'p44':letters, # fifth row
                   'p25':letters # sixth row
                   },
                  [Constraint(['p00', 'p10', 'p20'], is_word), #1-across
                   Constraint(['p00', 'p01', 'p02', 'p03'], is_word), # 1-down
                   Constraint(['p02', 'p12', 'p22', 'p32'], is_word), # 3-across
                   Constraint(['p20', 'p21', 'p22', 'p23', 'p24', 'p25'], is_word), # 2-down
                   Constraint(['p24', 'p34', 'p44'], is_word) # 4-across
                   ])
```

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

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

cspExamples.py — (continued)

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

def n_queens(n):
    """returns a CSP for n-queens"""
    columns = list(range(n))
    return CSP(
               {'R'+str(i):columns for i in range(n)},
                [Constraint(['R'+str(i),'R'+str(j)], queens(i,j)) for i in range(n) for j in range(n) i

# try the CSP n_queens(8) in one of the solvers.
# What is the smallest n for which there is a solution?
```

Exercise 4.1 How many constraints does this representation of the n-queens problem produce? Can it be done with fewer constraints? Either explain why it can't be done with fewer constraints, or give a solution using fewer constraints.

Unit tests

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

cspExamples.py — (continued)

```
def test(CSP_solver, csp=csp1,
            solutions=[{'A': 1, 'B': 3, 'C': 4}, {'A': 2, 'B': 3, 'C': 4}]):
    """CSP_solver is a solver that takes a csp and returns a solution
    csp is a constraint satisfaction problem
    solutions is the list of all solutions to csp
    This tests whether the solution returned by CSP_solver is a solution.
    """
    print("Testing csp with",CSP_solver.__doc__)
    sol0 = CSP_solver(csp)
    print("Solution found:",sol0)
    assert sol0 in solutions, "Solution not correct for "+str(csp)
    print("Passed unit test")
```

Exercise 4.2 Modify *test* so that instead of taking in a list of solutions, it checks whether the returned solution actually is a solution.

Exercise 4.3 Propose a test that is appropriate for CSPs with no solutions. Assume that the test designer knows there are no solutions. Consider what a CSP solver should return if there are no solutions to the CSP.

Exercise 4.4 Write a unit test that checks whether all solutions (e.g., for the search algorithms that can return multiple solutions) are correct, and whether all solutions can be found.

4.2 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.

The first solver 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. It then constructs a search space that can be solved using the search methods of the previous chapter. In this search space:

- A node is a *variable* : *value* dictionary which does not violate any constraints (so that dictionaries that vilate any constratints are not added).

- An arc corresponds to an assignment of a value to the next variable. This assumes a static ordering; the next variable chosen to split does not depend on the context. If no variable ordering is given, this makes no attempt to choose a good ordering.

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

```
from cspProblem import CSP, Constraint
from searchProblem import Arc, Search_problem
from utilities import dict_union

class Search_from_CSP(Search_problem):
    """A search problem directly from the CSP.

    A node is a variable:value dictionary"""
    def __init__(self, csp, variable_order=None):
        self.csp=csp
        if variable_order:
            assert set(variable_order) == set(csp.variables)
            assert len(variable_order) == len(csp.variables)
            self.variables = variable_order
        else:
            self.variables = list(csp.variables)

    def is_goal(self, node):
```

```
        """returns whether the current node is a goal for the search
        """
        return len(node)==len(self.csp.variables)

    def start_node(self):
        """returns the start node for the search
        """
        return {}
```

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

cspSearch.py — (continued)

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

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

cspSearch.py — (continued)

```
from cspExamples import csp1,csp2,test, crossword1, crossword1d
from searchGeneric import Searcher

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

if __name__ == "__main__":
    test(dfs_solver)

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

```
#print(searcher3.search()) # get next solution
searcher4 = Searcher(Search_from_CSP(crossword1d))
#print(searcher4.search()) # get next solution (warning: slow)
```

Exercise 4.5 What would happen if we constructed the new assignment by assigning $node[var] = val$ (with side effects) instead of using dictionary union? Give an example of where this could give a wrong answer. How could the algorithm be changed to work with side effects? (Hint: think about what information needs to be in a node).

Exercise 4.6 Change neighbors so that it returns an iterator of values rather than a list. (Hint: use *yield*.)

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.

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

```
from display import Displayable

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

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

cspConsistency.py — (continued)

```
    def make_arc_consistent(self, orig_domains=None, to_do=None):
        """Makes this CSP arc-consistent using generalized arc consistency
        orig_domains is the original domains
        to_do is a set of (variable,constraint) pairs
        returns the reduced domains (an arc-consistent variable:domain dictionary)
        """
        if orig_domains is None:
            orig_domains = self.csp.domains
```

```
        if to_do is None:
            to_do = {(var, const) for const in self.csp.constraints
                     for var in const.scope}
        else:
            to_do = to_do.copy() # use a copy of to_do
        domains = orig_domains.copy()
        self.display(2,"Performing AC with domains", domains)
        while to_do:
            var, const = self.select_arc(to_do)
            self.display(3, "Processing arc (", var, ",", const, ")")
            other_vars = [ov for ov in const.scope if ov != var]
            new_domain = {val for val in domains[var]
                            if self.any_holds(domains, const, {var: val}, other_vars)}
            if new_domain != domains[var]:
                self.display(4, "Arc: (", var, ",", const, ") is inconsistent")
                self.display(3, "Domain pruned", "dom(", var, ") =", new_domain,
                                 " due to ", const)
                domains[var] = new_domain
                add_to_do = self.new_to_do(var, const) - to_do
                to_do |= add_to_do    # set union
                self.display(3, " adding", add_to_do if add_to_do else "nothing", "to to_do.")
            self.display(4, "Arc: (", var, ",", const, ") now consistent")
        self.display(2, "AC done. Reduced domains", domains)
        return domains

    def new_to_do(self, var, const):
        """returns new elements to be added to to_do after assigning
        variable var in constraint const.
        """
        return {(nvar, nconst) for nconst in self.csp.var_to_const[var]
                if nconst != const
                for nvar in nconst.scope
                if nvar != var}
```

The following selects an arc. Any element of *to_do* can be selected. The selected element needs to be removed from *to_do*. The default implementation just selects which ever element *pop* method for sets returns. A user interface could allow the user to select an arc. Alternatively a more sophisticated selection could be employed (or just a stack or a queue).

cspConsistency.py — (continued)

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

The value of new_domain is the subset of the domain of var that is consistemt with the assignment to the other variables. It might be easier to understand the following code, which treats unary (with no other variables in the constraint) and binary (with one other variables in the constraint) constraints as special cases (this can replace the assignment to `new_domain` in the above code):

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

any_holds is a recursive function that tries to finds an assignment of values to the other variables (*other_vars*) that satisfies constraint *const* given the assignment in *env*. The integer variable *ind* specifies which index to *other_vars* needs to be checked next. As soon as one assignment returns *True*, the algorithm returns *True*. Note that it has side effects with respect to *env*; it changes the values of the variables in *other_vars*. It should only be called when the side effects have no ill effects.

cspConsistency.py — (continued)

```
    def any_holds(self, domains, const, env, other_vars, ind=0):
        """returns True if Constraint const holds for an assignment
        that extends env with the variables in other_vars[ind:]
        env is a dictionary
        Warning: this has side effects and changes the elements of env
        """
        if ind == len(other_vars):
            return const.holds(env)
        else:
            var = other_vars[ind]
            for val in domains[var]:
                # env = dict_union(env,{var:val}) # no side effects!
                env[var] = val
                if self.any_holds(domains, const, env, other_vars, ind + 1):
                    return True
            return False
```

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)

```
    def solve_one(self, domains=None, to_do=None):
        """return a solution to the current CSP or False if there are no solutions
        to_do is the list of arcs to check
        """
        if domains is None:
            domains = self.csp.domains
        new_domains = self.make_arc_consistent(domains, to_do)
        if any(len(new_domains[var]) == 0 for var in domains):
            return False
        elif all(len(new_domains[var]) == 1 for var in domains):
            self.display(2, "solution:", {var: select(
                new_domains[var]) for var in new_domains})
            return {var: select(new_domains[var]) for var in domains}
        else:
            var = self.select_var(x for x in self.csp.variables if len(new_domains[x]) > 1)
            if var:
                dom1, dom2 = partition_domain(new_domains[var])
                self.display(3, "...splitting", var, "into", dom1, "and", dom2)
                new_doms1 = copy_with_assign(new_domains, var, dom1)
                new_doms2 = copy_with_assign(new_domains, var, dom2)
                to_do = self.new_to_do(var, None)
                self.display(3, " adding", to_do if to_do else "nothing", "to to_do.")
                return self.solve_one(new_doms1, to_do) or self.solve_one(new_doms2, to_do)

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

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

The domains are implemented as a dictionary that maps each variables to its domain. Assigning a value in Python has side effects which we want to avoid. *copy_with_assign* takes a copy of the domains dictionary, perhaps allowing for a new domain for a variable. It 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)

```
def copy_with_assign(domains, var=None, new_domain={True, False}):
    """create a copy of the domains with an assignment var=new_domain
    if var==None then it is just a copy.
    """
    newdoms = domains.copy()
    if var is not None:
        newdoms[var] = new_domain
```

```
    return newdoms
```

cspConsistency.py — (continued)

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

    This implementation just picks the first element.
    For many of the uses, which element is selected does not affect correctness,
    but may affect efficiency.
    """
    for e in iterable:
        return e # returns first element found
```

Exercise 4.7 Implement of *solve_all* that is like *solve_one* but returns the set of all solutions.

Exercise 4.8 Implement *solve_enum* that enumerates the solutions. It should use Python's *yield* (and perhaps *yield from*).

Unit test:

cspConsistency.py — (continued)

```
from cspExamples import test
def ac_solver(csp):
    "arc consistency (solve_one)"
    return Con_solver(csp).solve_one()
if __name__ == "__main__":
    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.

A node is domains dictionary.

cspConsistency.py — (continued)

```
from searchProblem import Arc, Search_problem

class Search_with_AC_from_CSP(Search_problem,Displayable):
    """A search problem with arc consistency and domain splitting

    A node is a CSP """
    def __init__(self, csp):
        self.cons = Con_solver(csp) #copy of the CSP
        self.domains = self.cons.make_arc_consistent()

    def is_goal(self, node):
        """node is a goal if all domains have 1 element"""
        return all(len(node[var])==1 for var in node)

```

```
    def start_node(self):
        return self.domains

    def neighbors(self,node):
        """returns the neighboring nodes of node.
        """
        neighs = []
        var = select(x for x in node if len(node[x])>1)
        if var:
            dom1, dom2 = partition_domain(node[var])
            self.display(2,"Splitting", var, "into", dom1, "and", dom2)
            to_do = self.cons.new_to_do(var,None)
            for dom in [dom1,dom2]:
                newdoms = copy_with_assign(node,var,dom)
                cons_doms = self.cons.make_arc_consistent(newdoms,to_do)
                if all(len(cons_doms[v])>0 for v in cons_doms):
                    # all domains are non-empty
                    neighs.append(Arc(node,cons_doms))
                else:
                    self.display(2,"...",var,"in",dom,"has no solution")
        return neighs
```

Exercise 4.9 When splitting a domain, this code splits the domain into half, approximately in half (without any effort to make a sensible choice). Does it work better to split one element from a domain?

Unit test:

cspConsistency.py — (continued)

```
from cspExamples import test
from searchGeneric import Searcher

def ac_search_solver(csp):
    """arc consistency (search interface)"""
    sol = Searcher(Search_with_AC_from_CSP(csp)).search()
    if sol:
        return {v:select(d) for (v,d) in sol.end().items()}

if __name__ == "__main__":
    test(ac_search_solver)
```

Testing:

cspConsistency.py — (continued)

```
from cspExamples import csp1, csp2, csp3, csp4, crossword1, crossword1d

## Test Solving CSPs with Arc consistency and domain splitting:
#Con_solver.max_display_level = 4 # display details of AC (0 turns off)
#Con_solver(csp1).solve_one()
#searcher1d = Searcher(Search_with_AC_from_CSP(csp1))
#print(searcher1d.search())
```

```
#Searcher.max_display_level = 2 # display search trace (0 turns off)
#searcher2c = Searcher(Search_with_AC_from_CSP(csp2))
#print(searcher2c.search())
#searcher3c = Searcher(Search_with_AC_from_CSP(crossword1))
#print(searcher3c.search())
#searcher5c = Searcher(Search_with_AC_from_CSP(crossword1d))
#print(searcher5c.search())
```

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

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

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

    A node is a variable:value dictionary"""
    def __init__(self, csp):
        self.csp = csp
        self.variables_to_select = {var for var in self.csp.variables
                                    if len(self.csp.domains[var]) > 1}
        # Create assignment and conflicts set
        self.current_assignment = None # this will trigger a random restart
        self.number_of_steps = 0 #number of steps after the initialization
```

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

cspSLS.py — (continued)

```
    def restart(self):
        """creates a new total assignment and the conflict set
        """
        self.current_assignment = {var:random_sample(dom) for
                                   (var,dom) in self.csp.domains.items()}
        self.display(2,"Initial assignment",self.current_assignment)
        self.conflicts = set()
        for con in self.csp.constraints:
            if not con.holds(self.current_assignment):
                self.conflicts.add(con)
        self.display(2,"Number of conflicts",len(self.conflicts))
        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. The argument *pob_best* is the probability of selecting a best variable (one involving the most conflicts). When the value of *prob_best* is positive, the algorithm needs to maintain a priority queue of variables and the number of conflicts (using *search_with_var_pq*). If the probability of selecting a best variable is zero, it does not need to maintain this priority queue (as implemented in *search_with_any_conflict*).

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

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

cspSLS.py — (continued)

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

        max_steps is the maximum number of steps it will try before giving up
        prob_best is the probability that a best varaible (one in most conflict) is selected
        prob_anycon is the probability that a variabe in any conflict is selected
        (otherwise a variable is chosen at random)
        """
        if self.current_assignment is None:
            self.restart()
```

```
            self.number_of_steps += 1
            if not self.conflicts:
                self.display(1,"Solution found:", self.current_assignment, "after restart")
                return self.number_of_steps
        if prob_best > 0: # we need to maintain a variable priority queue
            return self.search_with_var_pq(max_steps, prob_best, prob_anycon)
        else:
            return self.search_with_any_conflict(max_steps, prob_anycon)
```

Exercise 4.10 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 keeps track of which variables are in conflicts.

cspSLS.py — (continued)

```
    def search_with_any_conflict(self, max_steps, prob_anycon=1.0):
        """Searches with the any_conflict heuristic.
        This relies on just maintaining the set of conflicts;
        it does not maintain a priority queue
        """
        self.variable_pq = None # we are not maintaining the priority queue.
                                # This ensures it is regenerated if
                                #   we call search_with_var_pq.
        for i in range(max_steps):
            self.number_of_steps +=1
            if  random.random() < prob_anycon:
                con = random_sample(self.conflicts) # pick random conflict
                var = random_sample(con.scope) # pick variable in conflict
            else:
                var = random_sample(self.variables_to_select)
            if len(self.csp.domains[var]) > 1:
                val = random_sample(self.csp.domains[var] -
                                    {self.current_assignment[var]})
                self.display(2,self.number_of_steps,": Assigning",var,"=",val)
                self.current_assignment[var]=val
                for varcon in self.csp.var_to_const[var]:
                    if varcon.holds(self.current_assignment):
                        if varcon in self.conflicts:
                            self.conflicts.remove(varcon)
                    else:
                        if varcon not in self.conflicts:
                            self.conflicts.add(varcon)
                self.display(2,"   Number of conflicts",len(self.conflicts))
            if not self.conflicts:
```

```
            self.display(1,"Solution found:", self.current_assignment,
                         "in", self.number_of_steps,"steps")
            return self.number_of_steps
    self.display(1,"No solution in",self.number_of_steps,"steps",
            len(self.conflicts),"conflicts remain")
    return None
```

Exercise 4.11 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 the 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 72) to find a variable with the most conflicts.

cspSLS.py — (continued)

```
def search_with_var_pq(self,max_steps, prob_best=1.0, prob_anycon=1.0):
    """search with a priority queue of variables.
    This is used to select a variable with the most conflicts.
    """
    if not self.variable_pq:
        self.create_pq()
    pick_best_or_con = prob_best + prob_anycon
    for i in range(max_steps):
        self.number_of_steps +=1
        randnum = random.random()
        ## Pick a variable
        if randnum < prob_best: # pick best variable
            var,oldval = self.variable_pq.top()
        elif randnum < pick_best_or_con: # pick a variable in a conflict
            con = random_sample(self.conflicts)
            var = random_sample(con.scope)
        else: #pick any variable that can be selected
            var = random_sample(self.variables_to_select)
        if len(self.csp.domains[var]) > 1: # var has other values
            ## Pick a value
            val = random_sample(self.csp.domains[var] -
                              {self.current_assignment[var]})
            self.display(2,"Assigning",var,val)
            ## Update the priority queue
            var_differential = {}
```

```
                self.current_assignment[var]=val
                for varcon in self.csp.var_to_const[var]:
                    self.display(3,"Checking",varcon)
                    if varcon.holds(self.current_assignment):
                        if varcon in self.conflicts: #was incons, now consis
                            self.display(3,"Became consistent",varcon)
                            self.conflicts.remove(varcon)
                            for v in varcon.scope: # v is in one fewer conflicts
                                var_differential[v] = var_differential.get(v,0)-1
                    else:
                        if varcon not in self.conflicts: # was consis, not now
                            self.display(3,"Became inconsistent",varcon)
                            self.conflicts.add(varcon)
                            for v in varcon.scope: # v is in one more conflicts
                                var_differential[v] = var_differential.get(v,0)+1
                self.variable_pq.update_each_priority(var_differential)
                self.display(2,"Number of conflicts",len(self.conflicts))
            if not self.conflicts: # no conflicts, so solution found
                self.display(1,"Solution found:", self.current_assignment,"in",
                             self.number_of_steps,"steps")
                return self.number_of_steps
        self.display(1,"No solution in",self.number_of_steps,"steps",
                    len(self.conflicts),"conflicts remain")
        return None
```

create_pq creates an updatable priority queue of the variables, ordered by the number of conflicts they participate in. The priority queue only includes variables in conflicts and the value of a variable is the *negative* of the number of conflicts the variable is in. This ensures that the priority queue, which picks the minimum value, picks a variable with the most conflicts.

cspSLS.py — (continued)

```
    def create_pq(self):
        """Create the variable to number-of-conflicts priority queue.
        This is needed to select the variable in the most conflicts.

        The value of a variable in the priority queue is the negative of the
        number of conflicts the variable appears in.
        """
        self.variable_pq = Updatable_priority_queue()
        var_to_number_conflicts = {}
        for con in self.conflicts:
            for var in con.scope:
                var_to_number_conflicts[var] = var_to_number_conflicts.get(var,0)+1
        for var,num in var_to_number_conflicts.items():
            if num>0:
                self.variable_pq.add(var,-num)
```

cspSLS.py — (continued)

```
def random_sample(st):
```

```
    """selects a random element from set st"""
    return random.sample(st,1)[0]
```

Exercise 4.12 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 the reduces the number of conflicts by the most. Have a parameter that specifies the probability that the best value is chosen.

Exercise 4.13 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)

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

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

    def add(self,elt,val):
        """adds elt to the priority queue with priority=val.
        """
        assert val <= 0,val
        assert elt not in self.elt_map, elt
```

```
        new_triple = [val, random.random(),elt]
        heapq.heappush(self.pq, new_triple)
        self.elt_map[elt] = new_triple

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

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

    def pop(self):
        """Removes and returns the (elt,value) pair with minimal value.
        If the priority queue is empty, IndexError is raised.
        """
        self.max_size = max(self.max_size, len(self.pq)) # keep statistics
        triple = heapq.heappop(self.pq)
        while triple[2] == self.REMOVED:
            triple = heapq.heappop(self.pq)
        del self.elt_map[triple[2]]
        return triple[2], triple[0] # elt, value

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

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

4.4.4 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)

```python
import matplotlib.pyplot as plt

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

    def plot_runs(self,num_runs=100,max_steps=1000, prob_best=1.0, prob_anycon=1.0):
        """Plots num_runs of SLS for the given settings.
        """
        stats = []
        SLSearcher.max_display_level, temp_mdl = 0, SLSearcher.max_display_level # no display
        for i in range(num_runs):
            searcher = SLSearcher(self.csp)
            num_steps = searcher.search(max_steps, prob_best, prob_anycon)
            if num_steps:
                stats.append(num_steps)
        stats.sort()
        if prob_best >= 1.0:
            label = "P(best)=1.0"
        else:
            p_ac = min(prob_anycon, 1-prob_best)
            label = "P(best)=%.2f, P(ac)=%.2f" % (prob_best, p_ac)
        plt.plot(stats,range(len(stats)),label=label)
        plt.legend(loc="upper left")
        #plt.draw()
        SLSearcher.max_display_level= temp_mdl #restore display
```

4.4.5 Testing

cspSLS.py — (continued)

```python
from cspExamples import test
def sls_solver(csp,prob_best=0.7):
```

```
    """stochastic local searcher (prob_best=0.7)"""
    se0 = SLSearcher(csp)
    se0.search(1000,prob_best)
    return se0.current_assignment
def any_conflict_solver(csp):
    """stochastic local searcher (any-conflict)"""
    return sls_solver(csp,0)

if __name__ == "__main__":
    test(sls_solver)
    test(any_conflict_solver)

from cspExamples import csp1, csp2, crossword1

## Test Solving CSPs with Search:
#se1 = SLSearcher(csp1); print(se1.search(100))
#se2 = SLSearcher(csp2); print(se2.search(1000,1.0)) # greedy
#se2 = SLSearcher(csp2); print(se2.search(1000,0)) # any_conflict
#se2 = SLSearcher(csp2); print(se2.search(1000,0.7)) # 70% greedy; 30% any_conflict
#SLSearcher.max_display_level=2 #more detailed display
#se3 = SLSearcher(crossword1); print(se3.search(100),0.7)
#p = Runtime_distribution(csp2)
#p.plot_runs(1000,1000,0) # any_conflict
#p.plot_runs(1000,1000,1.0) # greedy
#p.plot_runs(1000,1000,0.7) # 70% greedy; 30% any_conflict
```

Exercise 4.14 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 runtime, so you will be able to tell if there is a problem with the algorithm stopping.

Chapter 5

Propositions and Inference

5.1 Representing Knowledge Bases

A clause consists of a head (an atom) and a body. A body is represented as a list of atoms. Atoms are represented as strings.

logicProblem.py — Representations Logics

```
class Clause(object):
    """A definite clause"""

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

    def __str__(self):
        """returns the string representation of a clause.
        """
        if self.body:
            return self.head + " <- " + " & ".join(self.body) + "."
        else:
            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)

```
class Askable(object):
    """An askable atom"""

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

```
        self.atom=atom

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

def yes(ans):
    """returns true if the answer is yes in some form"""
    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.

logicProblem.py — (continued)

```
from display import Displayable

class KB(Displayable):
    """A knowledge base consists of a set of clauses.
    This also creates a dictionary to give fast access to the clauses with an atom in head.
    """
    def __init__(self, statements=[]):
        self.statements = statements
        self.clauses = [c for c in statements if isinstance(c, Clause)]
        self.askables = [c.atom for c in statements if isinstance(c, Askable)]
        self.atom_to_clauses = {} # dictionary giving clauses with atom as head
        for c in self.clauses:
            if c.head in self.atom_to_clauses:
                self.atom_to_clauses[c.head].add(c)
            else:
                self.atom_to_clauses[c.head] = {c}

    def clauses_for_atom(self,a):
        """returns set of clauses with atom a as the head"""
        if a in self.atom_to_clauses:
            return self.atom_to_clauses[a]
        else:
            return set()

    def __str__(self):
        """returns a string representation of this knowledge base.
        """
        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:

logicProblem.py — (continued)

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

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

logicProblem.py — (continued)

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

# print(kb)
```

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

```
from logicProblem import yes

def fixed_point(kb):
    """Returns the fixed point of knowledge base kb.
    """
    fp = ask_askables(kb)
    added = True
    while added:
        added = False  # added is true when an atom was added to fp this iteration
```

```
        for c in kb.clauses:
            if c.head not in fp and all(b in fp for b in c.body):
                fp.add(c.head)
                added = True
                kb.display(2,c.head,"added to fp due to clause",c)
    return fp

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

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

logicBottomUp.py — (continued)

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

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

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

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 asymptitocally more efficient (in terms of the number of elements in a body) than the method in the previous question by noticing that each element of the body of clause only needs to be checked once. For example, the clause $a \leftarrow b \wedge c \wedge d$, needs only be considered when b is added to fp. Once b is added to fp, if c is already in 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 displaying the code (and doesn't need to have a non-default value).

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

```
from logicProblem import yes

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

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

logicTopDown.py — (continued)

```
from logicProblem import triv_KB
def test():
    a1 = prove(triv_KB,['i_am'])
    assert a1, "triv_KB proving i_am gave "+str(a1)
    a2 = prove(triv_KB,['i_smell'])
    assert not a2, "triv_KB proving i_smell gave "+str(a2it)
    print("Passed unit tests")
if __name__ == "__main__":
    test()
# try
from logicProblem import elect
# elect.max_display_level=3 # give detailed trace
# prove(elect,['live_w6'])
# prove(elect,['lit_l1'])
```

Exercise 5.4 This code can re-ask a question multiple times. Implement this code so that it only asks a question once and remembers the answer. Also implement a function to forget the answers.

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

5.4 Assumables

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

logicAssumables.py — Definite clauses with assumables

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

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

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

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

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

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

logicAssumables.py — (continued)

```
    def prove_all_ass(self, ans_body, assumed=set()):
        """returns a list of sets of assumables that extends assumed
        to imply ans_body from self.
        ans_body is a list of atoms (it is the body of the answer clause).
        assumed is a set of assumables already assumed
        """
        if ans_body:
            selected = ans_body[0] # select first atom from ans_body
            if selected in self.askables:
                if yes(input("Is "+selected+" true? ")):
                    return self.prove_all_ass(ans_body[1:],assumed)
                else:
                    return []  # no answers
            elif selected in self.assumables:
                return self.prove_all_ass(ans_body[1:],assumed|{selected})
            else:
                return [ass
                        for cl in self.clauses_for_atom(selected)
```

```
                        for ass in self.prove_all_ass(cl.body+ans_body[1:],assumed)
                           ]  # union of answers for each clause with head=selected
        else:                 # empty body
            return [assumed]  # one answer

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

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

logicAssumables.py — (continued)

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

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

Warning: *minsets* works for a list of sets or for a set of (frozen) sets, but it does not work for a generator of sets. For example, try to predict and then test:

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

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

logicAssumables.py — (continued)

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

Test cases:

logicAssumables.py — (continued)

```
electa = KBA([
    Clause('light_l1'),
    Clause('light_l2'),
    Assumable('ok_l1'),
    Assumable('ok_l2'),
```

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

Exercise 5.6 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 5.7 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 5.8 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.

Chapter 6

Planning with Certainty

6.1 Representing Actions and Planning Problems

The STRIPS representation of an action consists of:

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

stripsProblem.py — STRIPS Representations of Actions

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

```
        self.preconds = preconds
        self.effects = effects
        self.cost = cost

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

A STRIPS domain consists of:

- A set of actions.
- A dictionary that maps each feature into a set of possible values for the feature.
- A list of the actions

stripsProblem.py — (continued)

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

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

stripsProblem.py — (continued)

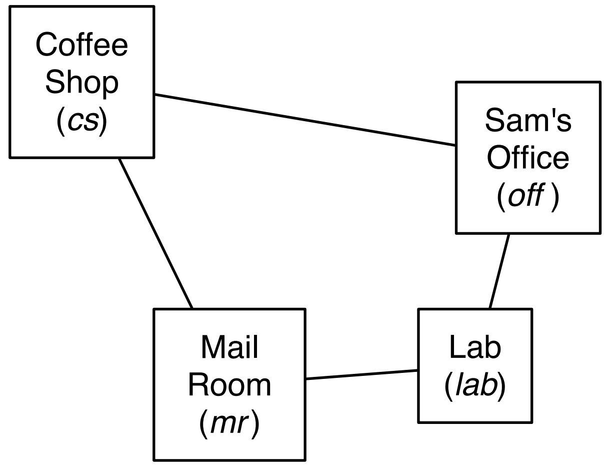
```
class Planning_problem(object):
    def __init__(self, prob_domain, initial_state, goal):
        """
        a planning problem consists of
        * a planning domain
        * the initial state
        * a goal
        """
        self.prob_domain = prob_domain
        self.initial_state = initial_state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

6.1.1 Robot Delivery Domain

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

stripsProblem.py — (continued)

```
boolean = {True, False}
```



Features to describe states

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

Actions

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

Figure 6.1: Robot Delivery Domain

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

stripsProblem.py — (continued)

```
problem0 = Planning_problem(delivery_domain,
                            {'RLoc':'lab', 'MW':True, 'SWC':True, 'RHC':False,
```

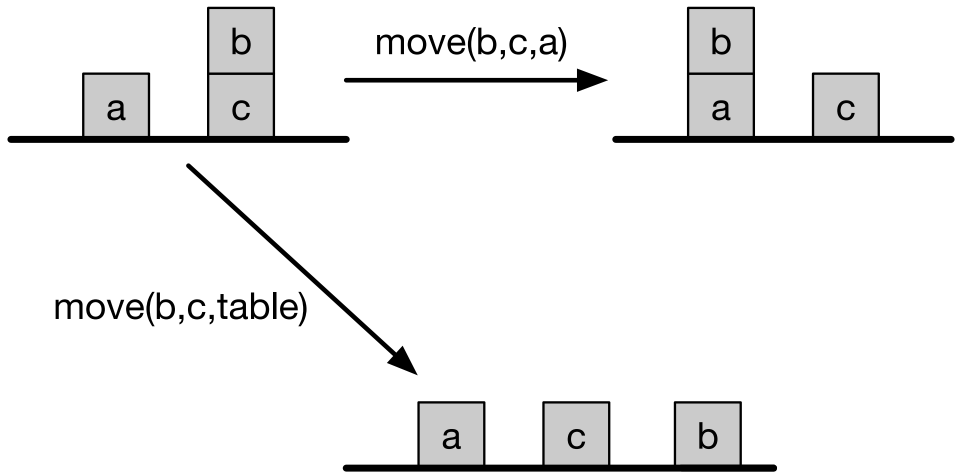


Figure 6.2: Blocks world with two actions

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

6.1.2 Blocks World

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

A state is defined by the two features:

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

There is one parameterized action

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

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

stripsProblem.py — (continued)

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

The problem *blocks*1 is a classic example, with 3 blocks, and the goal consists of two conditions. See Figure 6.3. Note that this example is challenging because we can't achieve one of the goals and then the other; whichever one we achieve first has to be undone to achieve the second.

stripsProblem.py — (continued)

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

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

stripsProblem.py — (continued)

```
blocks2dom = create_blocks_world({'a','b','c','d'})
tower4 = {clear('a'):True, on('a'):'b',
          clear('b'):False, on('b'):'c',
```

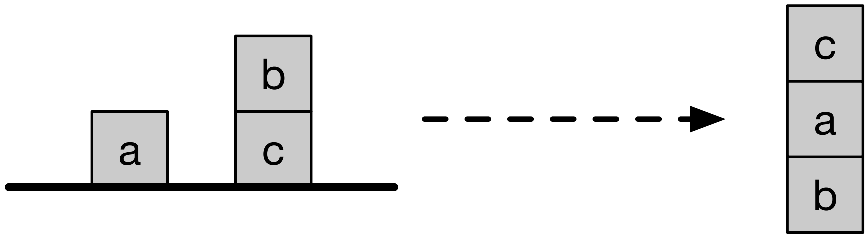


Figure 6.3: Blocks problem blocks1

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

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

stripsProblem.py — (continued)

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

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

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

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

6.2 Forward Planning

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

In a forward planner, a node is a state. A state consists of an assignment, 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

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

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

In order to define a search problem (page 33), 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)

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

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

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

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

```
                for prop in self.goal)

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

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

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

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

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

Here are some test cases to try.

stripsForwardPlanner.py — (continued)

```
from searchBranchAndBound import DF_branch_and_bound
from searchMPP import SearcherMPP
from stripsProblem import problem0, problem1, problem2, blocks1, blocks2, blocks3

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

6.2.1 Defining Heuristics for a Planner

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

Here is an example of defining a (not very good) heuristic for the coffee delivery planning domain.

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

stripsHeuristic.py — Planner with Heuristic Function

```
def dist(loc1, loc2):
    """returns the distance from location loc1 to loc2
    """
    if loc1==loc2:
        return 0
    if {loc1,loc2} in [{'cs','lab'},{'mr','off'}]:
        return 2
    else:
        return 1
```

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

stripsHeuristic.py — (continued)

```
def h1(state,goal):
    """ the distance to the goal location, if there is one"""
    if 'RLoc' in goal:
        return dist(state['RLoc'], goal['RLoc'])
    else:
        return 0

def h2(state,goal):
    """ the distance to the coffee shop plus getting coffee and delivering it
    if the robot needs to get coffee
    """
    if ('SWC' in goal and goal['SWC']==False
            and state['SWC']==True
            and state['RHC']==False):
        return dist(state['RLoc'],'cs')+3
    else:
        return 0
```

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

stripsHeuristic.py — (continued)

```
def maxh(*heuristics):
    """Returns a new heuristic function that is the maximum of the functions in heuristics.
    heuristics is the list of arguments which must be heuristic functions.
```

```
    """
    # return lambda state,goal: max(h(state,goal) for h in heuristics)
    def newh(state,goal):
        return max(h(state,goal) for h in heuristics)
    return newh
```

The following runs the example with and without the heuristic.

stripsHeuristic.py — (continued)

```
##### Forward Planner #####
from searchMPP import SearcherMPP
from stripsForwardPlanner import Forward_STRIPS
from stripsProblem import problem0, problem1, problem2, blocks1, blocks2, blocks3

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

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

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

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

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

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

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

i) $h3$ is like $h2$ but also takes into account the case when $Rloc$ is in goal.
ii) $h4$ uses the distance to the mail room plus getting mail and delivering it if the robot needs to get need to deliver mail.
iii) $h5$ is for getting mail when goal is for the robot to have mail, and then getting to the goal destination (if there is one).

Exercise 6.6 Create an admissible heuristic for the blocks world.

6.3 Regression Planning

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

In a regression planner a node is a subgoal that need to be achieved.

A *Subgoal* 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

```python
from searchProblem import Arc, Search_problem

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

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

stripsRegressionPlanner.py — (continued)

```python
from stripsForwardPlanner import zero

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

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

    def is_goal(self, subgoal):
        """if subgoal is true in the initial state, a path has been found"""
        goal_asst = subgoal.assignment
        return all(self.initial_state[g]==goal_asst[g]
                   for g in goal_asst)

```

```
    def start_node(self):
        """the start node is the top-level goal"""
        return self.top_goal

    def neighbors(self,subgoal):
        """returns a list of the arcs for the neighbors of subgoal in this problem"""
        goal_asst = subgoal.assignment
        return [ Arc(subgoal, self.weakest_precond(act,goal_asst), act.cost, act)
                 for act in self.prob_domain.actions
                 if self.possible(act,goal_asst)]

    def possible(self,act,goal_asst):
        """True if act is possible to achieve goal_asst.

        the action achieves an element of the effects and
        the action doesn't delete something that needs to be achieved and
        the preconditions are consistent with other subgoals that need to be achieved
        """
        return ( any(goal_asst[prop] == act.effects[prop]
                     for prop in act.effects if prop in goal_asst)
                and all(goal_asst[prop] == act.effects[prop]
                        for prop in act.effects if prop in goal_asst)
                and all(goal_asst[prop]== act.preconds[prop]
                        for prop in act.preconds if prop not in act.effects and prop in goal_asst)
                )

    def weakest_precond(self,act,goal_asst):
        """returns the subgoal that must be true so goal_asst holds after act
        should be:  act.preconds | (goal_asst - act.effects)
        """
        new_asst = act.preconds.copy()
        for g in goal_asst:
            if g not in act.effects:
                new_asst[g] = goal_asst[g]
        return Subgoal(new_asst)

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

stripsRegressionPlanner.py — (continued)

```
from searchBranchAndBound import DF_branch_and_bound
from searchMPP import SearcherMPP
from stripsProblem import problem0, problem1, problem2, blocks1, blocks2, blocks3

# SearcherMPP(Regression_STRIPS(problem1)).search() #A* with MPP
# DF_branch_and_bound(Regression_STRIPS(problem1),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 $\{a : \mathit{True}, b : \mathit{False}\}$ has been visited, then any node that is a superset, e.g., $\{a : \mathit{True}, b : \mathit{False}, d : \mathit{True}\}$, need not be expanded. If the simpler subgoal does not lead to a solution, the more complicated one wont 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, $\{'MW' : \mathit{True}, 'RHM' : \mathit{True}\}$ 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 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)

```
##### Regression Planner
from stripsRegressionPlanner import Regression_STRIPS

def test_regression_heuristic(thisproblem=problem1):
    print("\n***** REGRESSION NO HEURISTIC")
    print(SearcherMPP(Regression_STRIPS(thisproblem)).search())

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

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

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

Exercise 6.11 Create a 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

```
from cspProblem import CSP, Constraint

class CSP_from_STRIPS(CSP):
    """A CSP where:
    * a CSP variable is constructed by st(var,stage).
    * the dynamics are specified by the STRIPS representation of actions
    """

    def __init__(self, planning_problem, number_stages=2):
        prob_domain = planning_problem.prob_domain
        initial_state = planning_problem.initial_state
        goal = planning_problem.goal
        self.act_vars = [st('action',stage) for stage in range(number_stages)]
        domains = {av:prob_domain.actions for av in self.act_vars}
        domains.update({ st(var,stage):dom
                         for (var,dom) in prob_domain.feats_vals.items()
                         for stage in range(number_stages+1)})
        # initial state constraints:
        constraints = [Constraint((st(var,0),), is_(val))
                            for (var,val) in initial_state.items()]
        # goal constraints on the final state:
        constraints += [Constraint((st(var,number_stages),),
                                        is_(val))
                            for (var,val) in goal.items()]
        # precondition constraints:
        constraints += [Constraint((st(var,stage), st('action',stage)),
                                  if_(val,act)) # st(var,stage)==val if st('action',stage)=act
                            for act in prob_domain.actions
                            for var,val in act.preconds.items()
                            for stage in range(number_stages)]
        # effect constraints:
        constraints += [Constraint((st(var,stage+1), st('action',stage)),
                                  if_(val,act)) # st(var,stage+1)==val if st('action',stage)==act
```

```
                         for act in prob_domain.actions
                         for var,val in act.effects.items()
                         for stage in range(number_stages)]
        # frame constraints:
        constraints += [Constraint((st(var,stage), st('action',stage), st(var,stage+1)),
                                   eq_if_not_in_({act for act in prob_domain.actions
                                                  if var in act.effects}))
                         for var in prob_domain.feats_vals
                         for stage in range(number_stages) ]
        CSP.__init__(self, domains, constraints)

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

def st(var,stage):
    """returns a string for the var-stage pair that can be used as a variable"""
    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) returns *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)

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

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

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

```
    def eq_if_not_fun(x1, a, x2):
        return x1==x2 if a not in actset else True
    eq_if_not_fun.__name__ = "first and third arguments are equal if action is not in "+str(actset)
    return eq_if_not_fun
```

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)

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

The following are some example queries.

stripsCSPPlanner.py — (continued)

```
from searchGeneric import Searcher
from stripsProblem import delivery_domain
from cspConsistency import Search_with_AC_from_CSP, Con_solver
from stripsProblem import Planning_problem, problem0, problem1, problem2, blocks1, blocks2, blocks3

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

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

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

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

```

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

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

6.5 Partial-Order Planning

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

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

stripsPOP.py — Partial-order Planner using STRIPS representation

```
from searchProblem import Arc, Search_problem
import random

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

    def __str__(self):
        return str(self.action)+"#"+str(self.index)

    __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 $(a0, g, a1)$ triples, where a_1 and a_2 are action instances and g is a (var, val) pair. This holds when action a_0 makes g true for action a_1.

stripsPOP.py — (continued)

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

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

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

stripsPOP.py — (continued)

```
    def extract_plan(self):
        """returns a total ordering of the action instances consistent
        with the constraints.
        raises IndexError if there is no choice.
        """
        sorted_acts = []
        other_acts = set(self.actions)
        while other_acts:
            a = random.choice([a for a in other_acts if
                      all(((a1,a) not in self.constraints) for a1 in other_acts)])
```

```
            sorted_acts.append(a)
            other_acts.remove(a)
        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)

```
from display import Displayable

class POP_search_from_STRIPS(Search_problem, Displayable):
    def __init__(self,planning_problem):
        Search_problem.__init__(self)
        self.planning_problem = planning_problem
        self.start = Action_instance("start")
        self.finish = Action_instance("finish")

    def is_goal(self, node):
        return node.agenda == []

    def start_node(self):
        constraints = {(self.start, self.finish)}
        agenda = [(g, self.finish) for g in self.planning_problem.goal.items()]
        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)

```
    def neighbors(self, node):
        """enumerates the neighbors of node"""
        self.display(3,"finding neighbors of\n",node)
        if node.agenda:
            subgoal,act1 = node.agenda[0]
            self.display(2,"selecting",subgoal,"for",act1)
            new_agenda = node.agenda[1:]
            for act0 in node.actions:
                if (self.achieves(act0, subgoal) and
                   self.possible((act0,act1),node.constraints)):
                    self.display(2," reusing",act0)
                    consts1 = self.add_constraint((act0,act1),node.constraints)
                    new_clink = (act0,subgoal,act1)
                    new_cls = node.causal_links + [new_clink]
                    for consts2 in self.protect_cl_for_actions(node.actions,consts1,new_clink):
                        yield Arc(node,
                                  POP_node(node.actions,consts2,new_agenda,new_cls),
                                  cost=0)
            for a0 in self.planning_problem.prob_domain.actions: #a0 is an action
                if self.achieves(a0, subgoal):
                    #a0 acheieves subgoal
                    new_a = Action_instance(a0)
```

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

Given a 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)

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

```
    def protect_all_cls(self, clinks, act, constrs):
        """yields constraints that protect all causal links from act"""
        if clinks:
            (a0,cond,a1) = clinks[0] # select a causal link
```

```
                rem_clinks = clinks[1:] # remaining causal links
                if act != a0 and act != a1 and self.deletes(act,cond):
                    if self.possible((act,a0),constrs):
                        new_const = self.add_constraint((act,a0),constrs)
                        for e in self.protect_all_cls(rem_clinks,act,new_const): yield e
                    if self.possible((a1,act),constrs):
                        new_const = self.add_constraint((a1,act),constrs)
                        for e in self.protect_all_cls(rem_clinks,act,new_const): yield e
                else:
                    for e in self.protect_all_cls(rem_clinks,act,constrs): yield e
            else:
                yield constrs
```

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

stripsPOP.py — (continued)

```
    def achieves(self,action,subgoal):
        var,val = subgoal
        return var in self.effects(action) and self.effects(action)[var] == val

    def deletes(self,action,subgoal):
        var,val = subgoal
        return var in self.effects(action) and self.effects(action)[var] != val

    def effects(self,action):
        """returns the variable:value dictionary of the effects of action.
        works for both actions and action instances"""
        if isinstance(action, Action_instance):
            action = action.action
        if action == "start":
            return self.planning_problem.initial_state
        elif action == "finish":
            return {}
        else:
            return action.effects
```

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

stripsPOP.py — (continued)

```
    def add_constraint(self, pair, const):
        if pair in const:
            return const
        todo = [pair]
        newconst = const.copy()
        while todo:
            x0,x1 = todo.pop()
            newconst.add((x0,x1))
```

```
        for x,y in newconst:
            if x==x1 and (x0,y) not in newconst:
                todo.append((x0,y))
            if y==x0 and (x,x1) not in newconst:
                todo.append((x,x1))
    return newconst

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

Some code for testing:

stripsPOP.py — (continued)

```
from searchBranchAndBound import DF_branch_and_bound
from searchMPP import SearcherMPP
from stripsProblem import problem0, problem1, problem2, blocks1, blocks2, blocks3

rplanning0 = POP_search_from_STRIPS(problem0)
rplanning1 = POP_search_from_STRIPS(problem1)
rplanning2 = POP_search_from_STRIPS(problem2)
searcher0 = DF_branch_and_bound(rplanning0,5)
searcher0a = SearcherMPP(rplanning0)
searcher1 = DF_branch_and_bound(rplanning1,10)
searcher1a = SearcherMPP(rplanning1)
searcher2 = DF_branch_and_bound(rplanning2,10)
searcher2a = SearcherMPP(rplanning2)
# Try one of the following searchers
# a = searcher0.search()
# a = searcher0a.search()
# a.end().extract_plan() # print a plan found
# a.end().constraints  # print the constraints
# SearcherMPP.max_display_level = 0 # less detailed display
# DF_branch_and_bound.max_display_level = 0 # less detailed display
# a = searcher1.search()
# a = searcher1a.search()
# a = searcher2.search()
# a = searcher2a.search()
```

Chapter 7

Supervised Machine Learning

This chapter is the first on machine learning. It covers the following topics:

- Data: how to load it, training and test sets
- Features: many of the features come directly from the data. Sometimes it is useful to construct fratures, e.g. $height > 1.9m$ might be a Boolean feature constructed from the real-values feature *height*. The next chapter is about how to learn these features; in this chapter we construct them by hand, in what is often known a **feature engineering**.
- Learning with no input features: this is the base case of many methods. What should we predict if we have no input features? This is basic knowledge that everyone doing machine learning should know.
- Decision tree learning: ane of the classic and simplest learning algorithms, which is the basis of many other algorithms.
- Cross validations and parameter tuning: methods to prevent overfitting.
- Linear regression and classification: other classic and simple techniques that often work well (particularly combined with feature learning or engineering).
- Boosting: combining simpler learning methods to make even better learners.

A good source of classic datasets is the UCI machine Learning Repository [Lichman, 2013]. The SPECT and car datasets are from this repository.

7.1 Representations of Data and Predictions

The code uses the following definitions and conventions:

- A **data set** is an enumeration of examples.
- An **example** is a list (or tuple) of feature values. The feature values can be numbers or strings.
- A **feature** is a function from 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, if f is a Boolean feature, $f.frange == [False, True]$, and if e is an example, $f(e)$ is either *True* or *False*.

  The `__doc__` variable of the function contains the docstring, a string description of the function.

learnProblem.py — A Learning Problem

```
import math, random
import csv
from display import Displayable

boolean = [False, True]
```

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

learnProblem.py — (continued)

```
class Data_set(Displayable):
    """ A data set consists of a list of training data and a list of test data.
    """
    seed = None #123456 # make it None for a different test set each time

    def __init__(self, train, test=None, prob_test=0.30, target_index=0, header=None):
        """A dataset for learning.
        train is a list of tuples representing the training examples
        test is the list of tuples representing the test examples
        if test is None, a test set is created by selecting each
            example with probability prob_test
        target_index is the index of the target. If negative, it counts from right.
            If target_index is larger than the number of properties,
            there is no target (for unsupervised learning)
        header is a list of names for the features
        """
        if test is None:
            train,test = partition_data(train, prob_test, seed=self.seed)
        self.train = train
```

```
        self.test = test
        self.display(1,f"Training set has {len(train)} examples. Number of columns in ",{len(e) fo
        self.display(1,f"Test set has {len(test)} examples. Number of columns in ",{len(e) for e i
        self.prob_test = prob_test
        self.num_properties = len(self.train[0])
        if target_index < 0: #allows for -1, -2, etc.
            target_index = self.num_properties + target_index
        self.target_index = target_index
        self.header = header
        self.create_features()
        self.display(1,"There are",len(self.input_features),"input features")
```

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

learnProblem.py — (continued)

```
    def create_features(self):
        """create the input features and target feature.
        This assumes that the features all have range {0,1}.
        This should be overridden if the features have a different range.
        """
        self.input_features = []
        for i in range(self.num_properties):
            def feat(e,index=i):
                return e[index]
            if self.header:
                feat.__doc__ = self.header[i]
            else:
                feat.__doc__ = "e["+str(i)+"]"
            feat.frange = [0,1]
            if i == self.target_index:
                self.target = feat
            else:
                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.

An error measure takes a prediction and the actual value and returns a non-negative real number, such that the error for a dataset is the mean of the errors for each example. We assume that a lower error is better.

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 squared error (average of the square of the difference between the actual and predicted values), absolute errors(average of the absolute difference between the actual and predicted values) and the logloss (the a average negative log-likelihood, which can be interpreted as the number of bits to describe an example using a code based on the prediction treated as a probability).

learnProblem.py — (continued)

```
    def evaluate_dataset(self, data, predictor, error_measure):
        """Evaluates predictor on data according to the error_measure
        predictor is a function that takes an example and returns a
                prediction for the target feature.
        error_measure(prediction,actual) -> non-negative reals
        """
        if data:
            try:
                error = mean(error_measure(predictor(e), self.target(e))
                            for e in data)
            except ValueError:
                return float("inf") # infinity
            return error
```

The following evaluation criteria are defined. (Please keep the `__doc__` strings a consistent length as they are used in tables.)

learnProblem.py — (continued)

```
def squared_error(prediction,actual):
    "squared error "
    return (prediction-actual)**2
def absolute_error(prediction, actual):
    "absolute error"
    return abs(prediction-actual)
def log_loss(prediction,actual):
    "logloss     "
    try:
      if actual==0:
        return -math.log2(1-prediction)
      else:
        return -math.log2(prediction)
    except ValueError:
        return float("inf") # infinity

evaluation_criteria = [squared_error, absolute_error, log_loss]
```

The following computes the mean of an enumeration, with an optional initial sum and initial count. This works for enumerations, even where $len()$ is not defined, and only goes through the enumeration once. The obvious way to compute a mean: $sum(enum)/len(enum)$ works for lists, but does not work for arbitrary enumerations.

learnProblem.py — (continued)

```
def mean(enum, isum=0, icount=0):
    """returns the mean of enumeration enum,
       isum is the initial sum, and icount is the initial count."""
    for e in enum:
```

```
            icount += 1
            isum += e
    return isum/icount
```

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)

```
def partition_data(data, prob_test=0.30, seed=None):
    """partitions the data into a training set and a test set, where
    prob_test is the probability of each example being in the test set.
    """
    train = []
    test = []
    if seed:   # given seed makes the partition consistent from run-to-run
        random.seed(seed)
    for example in data:
        if random.random() < prob_test:
            test.append(example)
        else:
            train.append(example)
    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_all* 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; specified by *include_only*. Note that if *include_only* is specified, the target index is the column in the remaining columns.

learnProblem.py — (continued)

```
class Data_from_file(Data_set):
    def __init__(self, file_name, separator=',', num_train=None, prob_test=0.3,
                 has_header=False, target_index=0, boolean_features=True,
                 categorical=[], include_only=None):
        """create a dataset from a file
        separator is the character that separates the attributes
        num_train is a number n specifying the first n tuples are training, or None
        prob_test is the probability an example should in the test set (if num_train is None)
        has_header is True if the first line of file is a header
        target_index specifies which feature is the target
        boolean_features specifies whether we want to create Boolean features
            (if False, it uses the original features).
        categorical is a set (or list) of features that should be treated as categorical
        include_only is a list or set of indexes of columns to include
        """
        self.boolean_features = boolean_features
        with open(file_name,'r',newline='') as csvfile:
            # data_all = csv.reader(csvfile,delimiter=separator) # for more complicated CSV files
            data_all = (line.strip().split(separator) for line in csvfile)
            if include_only is not None:
                data_all = ([v for (i,v) in enumerate(line) if i in include_only]
                                for line in data_all)
            if has_header:
                header = next(data_all)
            else:
                header = None
            data_tuples = (interpret_elements(d) for d in data_all if len(d)>1)
            if num_train is not None:
                # training set is divided into training then text examples
                # the file is only read once, and the data is placed in appropriate list
                train = []
                for i in range(num_train):  # will give an error if insufficient examples
                    train.append(next(data_tuples))
                test = list(data_tuples)
                Data_set.__init__(self,train, test=test, target_index=target_index,header=header)
            else:    # randomly assign training and test examples
                Data_set.__init__(self,data_tuples, test=None, prob_test=prob_test,
                                  target_index=target_index, header=header)

    def __str__(self):
        if self.train and len(self.train)>0:
            return ("Data: "+str(len(self.train))+" training examples, "
                    +str(len(self.test))+" test examples, "
```

```
                +str(len(self.train[0]))+" features.")
        else:
            return ("Data: "+str(len(self.train))+" training examples, "
                    +str(len(self.test))+" test examples.")
```

7.1.4 Creating Binary Features

Some of the algorithms require Boolean features or features with range $\{0, 1\}$. In order to be able to use these algorithms 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 range 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) 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 a value that doesn't appear in the training set for that feature, the indicator functions all return false.

learnProblem.py — (continued)

```
    def create_features(self, max_num_cuts=8):
        """creates boolean features from input features.
        max_num_cuts is the maximum number of binary variables
           to split a numerical feature into.
        """
        ranges = [set() for i in range(self.num_properties)]
        for example in self.train:
            for ind,val in enumerate(example):
                ranges[ind].add(val)
        if self.target_index <= self.num_properties:
            # If target_index is larger than the number of properties,
            # there is no target (for unsupervised learning)
            def target(e,index=self.target_index):
                return e[index]
            if self.header:
                target.__doc__ = self.header[ind]
```

```
            else:
                target.__doc__ = "e["+str(ind)+"]"
            target.frange = ranges[self.target_index]
            self.target = target
        if self.boolean_features:
            self.input_features = []
            for ind,frange in enumerate(ranges):
                if ind != self.target_index and len(frange)>1:
                    if len(frange) == 2:
                        # two values, the feature is equality to one of them.
                        true_val = list(frange)[1] # choose one as true
                        def feat(e, i=ind, tv=true_val):
                            return e[i]==tv
                        if self.header:
                            feat.__doc__ = self.header[ind]+"=="+str(true_val)
                        else:
                            feat.__doc__ = "e["+str(ind)+"]=="+str(true_val)
                        feat.frange = boolean
                        self.input_features.append(feat)
                    elif all(isinstance(val,(int,float)) for val in frange):
                        # all numeric, create cuts of the data
                        sorted_frange = sorted(frange)
                        num_cuts = min(max_num_cuts,len(frange))
                        cut_positions = [len(frange)*i//num_cuts for i in range(1,num_cuts)]
                        for cut in cut_positions:
                            cutat = sorted_frange[cut]
                            def feat(e, ind_=ind, cutat=cutat):
                                return e[ind_] < cutat

                            if self.header:
                                feat.__doc__ = self.header[ind]+"<"+str(cutat)
                            else:
                                feat.__doc__ = "e["+str(ind)+"]<"+str(cutat)
                            feat.frange = boolean
                            self.input_features.append(feat)
                    else:
                        # create an indicator function for every value
                        for val in frange:
                            def feat(e, ind_=ind, val_=val):
                                return e[ind_] == val_
                            if self.header:
                                feat.__doc__ = self.header[ind]+"=="+str(val)
                            else:
                                feat.__doc__= "e["+str(ind)+"]=="+str(val)
                            feat.frange = boolean
                            self.input_features.append(feat)
        else: # boolean_features is off
            self.input_features = []
            for i in range(self.num_properties):
                def feat(e,index=i):
```

```
                    return e[index]
                if self.header:
                    feat.__doc__ = self.header[i]
                else:
                     feat.__doc__ = "e["+str(i)+"]"
                feat.frange = ranges[i]
                if i == self.target_index:
                    self.target = feat
                else:
                    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. As a test case, make sure that when the range is the 30 integers from 100 to 129, and you want 2 cuts, the resulting Boolean features should be $e[ind] \leq 109$ and $e[ind] \leq 119$ to make sure that each of the resulting 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) or Boolean, if it is possible.

learnProblem.py — (continued)

```
def interpret_elements(str_list):
    """make the elements of string list str_list numerical if possible.
    Otherwise remove initial and trailing spaces.
    """
    res = []
    for e in str_list:
        try:
            res.append(int(e))
        except ValueError:
            try:
                res.append(float(e))
            except ValueError:
                se = e.strip()
                if se in ["True","true","TRUE"]:
                    res.append[True]
                if se in ["False","false","FALSE"]:
                    res.append[False]
                else:
                    res.append(e.strip())
    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. Note that these are sometimes called **kernel**s; mapping the original feature space into a new space, from which we can use standard learning tools. For those interested in the mathematics, read about support vector machines, which have neat way to do learning in the augmented space (the "kernel trick") that is beyond the scope of AIPython (currently).

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)

```
class Data_set_augmented(Data_set):
    def __init__(self, dataset, unary_functions=[], binary_functions=[], include_orig=True):
        """creates a dataset like dataset but with new features
        unary_function is a list of unary feature constructors
        binary_functions is a list of binary feature combiners.
        include_orig specifies whether the original features should be included
        """
        self.orig_dataset = dataset
        self.unary_functions = unary_functions
        self.binary_functions = binary_functions
        self.include_orig = include_orig
        self.target = dataset.target
        Data_set.__init__(self,dataset.train, test=dataset.test,
                          target_index = dataset.target_index)

    def create_features(self):
        if self.include_orig:
            self.input_features = self.orig_dataset.input_features.copy()
        else:
            self.input_features = []
        for u in self.unary_functions:
            for f in self.orig_dataset.input_features:
                self.input_features.append(u(f))
        for b in self.binary_functions:
            for f1 in self.orig_dataset.input_features:
                for f2 in self.orig_dataset.input_features:
                    if f1 != f2:
                        self.input_features.append(b(f1,f2))
```

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

learnProblem.py — (continued)

```
def square(f):
    """a unary feature constructor to construct the square of a feature
```

```
    """
    def sq(e):
        return f(e)**2
    sq.__doc__ = f.__doc__+"**2"
    return sq

def power_feat(n):
    """given n returns a unary feature constructor to construct the nth power of a feature.
    e.g., power_feat(2) is the same as square, defined above
    """
    def fn(f,n=n):
        def pow(e,n=n):
            return f(e)**n
        pow.__doc__ = f.__doc__+"**"+str(n)
        return pow
    return fn

def prod_feat(f1,f2):
    """a new feature that is the product of features f1 and f2
    """
    def feat(e):
        return f1(e)*f2(e)
    feat.__doc__ = f1.__doc__+"*"+f2.__doc__
    return feat

def eq_feat(f1,f2):
    """a new feature that is 1 if f1 and f2 give same value
    """
    def feat(e):
        return 1 if f1(e)==f2(e) else 0
    feat.__doc__ = f1.__doc__+"=="+f2.__doc__
    return feat

def neq_feat(f1,f2):
    """a new feature that is 1 if f1 and f2 give different values
    """
    def feat(e):
        return 1 if f1(e)!=f2(e) else 0
    feat.__doc__ = f1.__doc__+"!="+f2.__doc__
    return feat
```

Example:

learnProblem.py — (continued)

```
# from learnProblem import Data_set_augmented,prod_feat
# data = Data_from_file('data/holiday.csv', num_train=19, target_index=-1)
## data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0)
# dataplus = Data_set_augmented(data,[],[prod_feat])
# dataplus = Data_set_augmented(data,[],[prod_feat,neq_feat])
```

Exercise 7.3 For symmetric properties, such as product, we don't need both

$f1 * f2$ as well as $f2 * f1$ as extra properties. Allow the user to be able to declare feature constructors as symmetric (by associating a Boolean feature with them). Change *construct_features* so that it does not create both versions for symmetric combiners.

7.2 Generic Learner Interface

A **learner** takes a dataset (and possibly other arguments specific to the method). To get it to learn, we call the *learn*() method. This implements *Displayable* so that we can display traces at multiple levels of detail (and perhaps with a GUI).

learnProblem.py — (continued)

```
from display import Displayable

class Learner(Displayable):
    def __init__(self, dataset):
        raise NotImplementedError("Learner.__init__") # abstract method

    def learn(self):
        """returns a predictor, a function from a tuple to a value for the target feature
        """
        raise NotImplementedError("learn") # abstract method
```

7.3 Learning With No Input Features

If we make the same prediction for each example, what prediction should we make? This can be used as a naive baseline; if a method does not do better than this, the the input features do not provide any useful information for the prediction. It is also the base case for some methods, such as decision-tree learning.

> To run demo to compare different prediction methods on various evaluation criteria, in folder "aipython", load "learnNoInputs.py", using e.g., `ipython -i learnNoInputs.py`, and it prints some test results.

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

- a point prediction, where 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 of 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 that 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 third can be implemented by having multiple indicator functions for the target.

Here are some prediction functions that take in a dataset of number and returns a prediction for the next case. Note that median will average the two middle values when there are an even number of examples. The mode will pick one of the values arbitrarily (here the larger) when more than one value has the maximum number of elements. So the median of $[0,1]$ is 0.5, but the mode is 1.

learnNoInputs.py — Learning ignoring all input features

```python
from learnProblem import squared_error, absolute_error, log_loss, mean
import math, random, statistics
import utilities # argmax for (element,value) pairs

class Predict(object):
    """The class of prediction methods for a list of numbers
    Please make the doc strings the same length, because they are used in tables.
    Note that we don't need self argument, as we are creating Predict objects,
    To use call Predict.laplace(data) etc."""

    def mean(data):
        "mean       "
        return mean(data)

    def bounded_mean(data, bound=0.01):
        "bounded mean"
        return min(max(mean(data),bound),1-bound)

    def laplace(data):
        "Laplace    "  # for Boolean (or 0/1 data only)
        return mean(data, isum=1, icount=2)

    def mode(data):
        "mode       "
        counts = {}
        for e in data:
            if e in counts:
                counts[e] += 1
            else:
```

```
                counts[e] = 1
        return utilities.argmaxe(counts.items())

    def median(data):
        "median    "
        return statistics.median(data)

    all = [mean, bounded_mean, laplace, mode, median]
```

7.3.1 Evaluation

To evaluate a point prediction, we first generate some data from a simple (Bernoulli) distribution, where there are two possible values, 0 and 1 for the target feature. 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. To generate a 1 with probability *prob*, we generarate a random number in range [0,1] and return 1 if that number is less than *prob*.

learnNoInputs.py — (continued)

```
def evaluate(train_size, predictor, error_measure, num_samples=10000, test_size=10 ):
    """return the average error when
   train_size is the number of training examples
   predictor(training) -> [0,1]
   error_measure(prediction,actual) -> non-negative reals
   """
    error = 0
    for sample in range(num_samples):
        prob = random.random()
        training = [1 if random.random()<prob else 0 for i in range(train_size)]
        prediction = predictor(training)
        test = (1 if random.random()<prob else 0 for i in range(test_size))
        error += sum( error_measure(prediction,actual) for actual in test)/test_size
    return error/num_samples
```

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

learnNoInputs.py — (continued)

```
def test_no_inputs(error_measures = [squared_error, absolute_error, log_loss]):
    for train_size in [1,2,3,4,5,10,20,100,1000]:
        print("For training size",train_size,":")
        print("  Predictor","\t".join(error_measure.__doc__ for
                                      error_measure in error_measures),sep="\t")
        for predictor in Predict.all:
            print(f"  {predictor.__doc__}",
                      "\t".join("{:.7f}".format(evaluate(train_size, predictor, error_measure))
                                 for error_measure in error_measures),sep="\t")

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

Exercise 7.4 Wich predictor works best for low counts when the error is

(a) Squared error
(b) Absolute error
(c) Log loss

You may need to try this a few times to make sure your answer is supported by the evidence. Does the difference from the other methods get more ot less as the number of examples grow?

Exercise 7.5 Suggest some other predictions that only take the tarining data. Does your method do better than the given methods? A simple way to get other predictors is to vary the threshold of bounded average, or to change the pseodo-counts of the Laplace method (use other numbers instead of 1 and 2).

7.4 Decision Tree Learning

> To run the decision tree learning demo, in folder "`aipython`", load "`learnDT.py`", using e.g., `ipython -i learnDT.py`, and it prints some test results. To try more examples, copy and paste the commented-out commands at the bottom of that file. This requires Python 3 with `matplotlib`.

The decision tree algorithm does binary splits, and assumes that all input features are binary functions of the examples. 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 feature.

learnDT.py — Learning a binary decision tree

```
from learnProblem import Learner, squared_error, absolute_error, log_loss, mean
from learnNoInputs import Predict
import math

class DT_learner(Learner):
    def __init__(self,
                 dataset,
                 split_to_optimize=log_loss,       # to minimize for at each split
                 leaf_prediction=Predict.mean, # what to use for point prediction at leaves
                 train=None,                  # used for cross validation
                 min_number_examples=10):
        self.dataset = dataset
        self.target = dataset.target
        self.split_to_optimize = split_to_optimize
        self.leaf_prediction = leaf_prediction
        self.min_number_examples = min_number_examples
        if train is None:
            self.train = self.dataset.train
        else:
            self.train = train
```

```
    def learn(self):
        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 splits unless:

- there are no more input features
- there are fewer examples than *min_number_examples*,
- all the examples agree on the value of the target, or
- the best split makes all examples in the same partition.

If it splits, it selects the best split according to the evaluation critereon (assuming that is the only split it gets to do), and returns the condition to split on (in the variable *split*) and the corresponding partition of the examples.

learnDT.py — (continued)

```
    def learn_tree(self, input_features, data_subset):
        """returns a decision tree
        for input_features is a set of possible conditions
        data_subset is a subset of the data used to build this (sub)tree

        where a decision tree is a function that takes an example and
        makes a prediction on the target feature
        """
        if (input_features and len(data_subset) >= self.min_number_examples):
            first_target_val = self.target(data_subset[0])
            allagree = all(self.target(inst)==first_target_val for inst in data_subset)
            if not allagree:
                split, partn = self.select_split(input_features, data_subset)
                if split: # the split succeeded in splitting the data
                    false_examples, true_examples = partn
                    rem_features = [fe for fe in input_features if fe != split]
                    self.display(2,"Splitting on",split.__doc__,"with examples split",
                                 len(true_examples),":",len(false_examples))
                    true_tree = self.learn_tree(rem_features,true_examples)
                    false_tree = self.learn_tree(rem_features,false_examples)
                    def fun(e):
                        if split(e):
                            return true_tree(e)
                        else:
                            return false_tree(e)
                    #fun = lambda e: true_tree(e) if split(e) else false_tree(e)
                    fun.__doc__ = ("if "+split.__doc__+" then ("+true_tree.__doc__+
                                 ") else ("+false_tree.__doc__+")")
                    return fun
```

```
        # don't expand the trees but return a point prediction
        prediction = self.leaf_prediction(self.target(e) for e in data_subset)
        def leaf_fun(e):
            return prediction
        leaf_fun.__doc__ = "{:.7f}".format(prediction)
        return leaf_fun
```

learnDT.py — (continued)

```
    def select_split(self, input_features, data_subset):
        """finds best feature to split on.

        input_features is a non-empty list of features.
        returns feature, partition
        where feature is an input feature with the smallest error as
              judged by split_to_optimize or
              feature==None if there are no splits that improve the error
        partition is a pair (false_examples, true_examples) if feature is not None
        """
        best_feat = None # best feature
        # best_error = float("inf") # infinity - more than any error
        best_error = training_error(self.dataset.target, data_subset,
                                        self.split_to_optimize, self.leaf_prediction)
        best_partition = None
        for feat in input_features:
            false_examples, true_examples = partition(data_subset,feat)
            if false_examples and true_examples: #both partitons are non-empty
                err = (training_error(self.dataset.target, false_examples,
                                          self.split_to_optimize, self.leaf_prediction)
                       + training_error(self.dataset.target, true_examples,
                                            self.split_to_optimize, self.leaf_prediction))
                self.display(3," split on",feat.__doc__,"has error=",err,
                          "splits into",len(true_examples),":",len(false_examples))
                if err < best_error:
                    best_feat = feat
                    best_error=err
                    best_partition = false_examples, true_examples
        self.display(3,"best split is on",best_feat.__doc__,
                               "with err=",best_error)
        return best_feat, best_partition

def partition(data_subset,feature):
    """partitions the data_subset by the feature"""
    true_examples = []
    false_examples = []
    for example in data_subset:
        if feature(example):
            true_examples.append(example)
        else:
            false_examples.append(example)
    return false_examples, true_examples
```

```python


def training_error(target, data_subset, eval_critereon, leaf_prediction):
    """returns training error for dataset on the target (with no more splits)
    We make a single prediction using leaf_prediction
    It is evaluated using eval_critereon for each example
    """
    prediction = leaf_prediction(target(e) for e in data_subset)
    error = sum(eval_critereon(prediction, target(e))
                for e in data_subset)
    return error
```

Test cases:

learnDT.py — (continued)

```python
from learnProblem import Data_set, Data_from_file

def testDT(data, print_tree=True, selections = Predict.all):
    """Prints errors and the trees for various evaluation criteria and ways to select leaves.
    """
    evaluation_criteria = [squared_error, absolute_error, log_loss]
    print("Split Choice","Leaf Choice",'\t'.join(ecrit.__doc__
                                               for ecrit in evaluation_criteria),sep="\t")
    for crit in evaluation_criteria:
        for leaf in selections:
            tree = DT_learner(data, split_to_optimize=crit, leaf_prediction=leaf).learn()
            print(crit.__doc__, leaf.__doc__,
                  "\t".join("{:.7f}".format(data.evaluate_dataset(data.test, tree, ecrit))
                            for ecrit in evaluation_criteria),sep="\t")
            if print_tree:
                print(tree.__doc__)

if __name__ == "__main__":
    print("SPECT.csv"); testDT(data=Data_from_file('data/SPECT.csv', target_index=0), print_tree=False
    # print("carbool.csv"); testDT(data = Data_from_file('data/carbool.csv', target_index=-1))
    # print("mail_reading.csv"); testDT(data = Data_from_file('data/mail_reading.csv', target_index=-
    # print("holiday.csv"); testDT(data = Data_from_file('data/holiday.csv', num_train=19, target_inde
```

Note that different runs may provide different values as they splt the training and test sets differetly. So if you have a hypothesis about what works better, make sure it is true for different runs.

Exercise 7.6 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 7.7 Extend the current algorithm to include in the stopping criterion

(a) A minimum child size; don't use a split if one of the children has fewer elements that this.

(b) A depth-bound on the depth of the tree.

(c) An improvement bound such that a split is only carried out if error with the split is better than the error without the split by at least the improvement bound.

Which values for these parameters make the prediction errors on the test set the smallest? Try it on more than one dataset.

Exercise 7.8 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 7.9 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?

7.5 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 the examples at the end 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

```
from learnProblem import Data_set, Data_from_file, squared_error, absolute_error, log_loss
from learnDT import DT_learner
import matplotlib.pyplot as plt
import random

class K_fold_dataset(object):
    def __init__(self, training_set, num_folds):
        self.data = training_set.train.copy()
        self.target = training_set.target
        self.input_features = training_set.input_features
        self.num_folds = num_folds
        random.shuffle(self.data)
        self.fold_boundaries = [(len(self.data)*i)//num_folds
                                for i in range(0,num_folds+1)]

    def fold(self, fold_num):
        for i in range(self.fold_boundaries[fold_num],
                       self.fold_boundaries[fold_num+1]):
            yield self.data[i]

    def fold_complement(self, fold_num):
        for i in range(0,self.fold_boundaries[fold_num]):
            yield self.data[i]
        for i in range(self.fold_boundaries[fold_num+1],len(self.data)):
            yield self.data[i]
```

The validation error is the average error for each example, where we test on each fold, and learn on the other folds.

learnCrossValidation.py — (continued)

```
    def validation_error(self, learner, error_measure, **other_params):
        error = 0
        try:
            for i in range(self.num_folds):
                predictor = learner(self, train=list(self.fold_complement(i)),
                                    **other_params).learn()
                error += sum( error_measure(predictor(e), self.target(e))
                              for e in self.fold(i))
        except ValueError:
            return float("inf") #infinity
        return error/len(self.data)
```

The *plot_error* method plots the average error as a function of a the minimun 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 is were to be used this way then it cannot be used to test.

learnCrossValidation.py — (continued)

```
def plot_error(data, criterion=squared_error, num_folds=5, xscale='linear'):
    """Plots the error on the validation set and the test set
    with respect to settings of the minimum number of examples.
    xscale should be 'log' or 'linear'
    """
    plt.ion()
    plt.xscale(xscale) # change between log and linear scale
    plt.xlabel("minimum number of examples")
    plt.ylabel("average "+criterion.__doc__)
    folded_data = K_fold_dataset(data, num_folds)
    verrors = []  # validation errors
    terrors = []  # test set errors
    for mne in range(1,len(data.train)+2):
        verrors.append(folded_data.validation_error(DT_learner,criterion,
                                            min_number_examples=mne))
        tree = DT_learner(data, criterion, min_number_examples=mne).learn()
        terrors.append(data.evaluate_dataset(data.test,tree,criterion))
    plt.plot(range(1,len(data.train)+2), verrors, ls='-',color='k',
                 label="validation for "+criterion.__doc__)
    plt.plot(range(1,len(data.train)+2), terrors, ls='--',color='k',
                 label="test set for "+criterion.__doc__)
    plt.legend()
    plt.draw()

# The following produces Figure 7.15 of Poole and Mackworth [2017]
# Different runs produce different plots, because folds change.
# data = Data_from_file('data/SPECT.csv',target_index=0)
# plot_error(data)  # warning, may take a long time depending on the dataset

#also try:
# data = Data_from_file('data/mail_reading.csv', target_index=-1)
# data = Data_from_file('data/carbool.csv', target_index=-1)
```

Note that different runs for the same data will have the same test error, but different validation error. If you rerun the `Data_from_file`, you will get the new test and training sets, and so the graph will change.

Exercise 7.10 Change the error plot so that it can evaluate the stopping criteria of the exercise of Section 7.6. Which criteria makes the most difference?

7.6 Linear Regression and Classification

Here we give a gradient descent searcher for linear regression and classification.

learnLinear.py — Linear Regression and Classification

```
from learnProblem import Learner
import random, math

class Linear_learner(Learner):
```

```
    def __init__(self, dataset, train=None,
                 learning_rate=0.1, max_init = 0.2,
                 squashed=True):
        """Creates a gradient descent searcher for a linear classifier.
        The main learning is carried out by learn()

        dataset provides the target and the input features
        train provides a subset of the training data to use
        number_iterations is the default number of steps of gradient descent
        learning_rate is the gradient descent step size
        max_init is the maximum absolute value of the initial weights
        squashed specifies whether the output is a squashed linear function
        """
        self.dataset = dataset
        self.target = dataset.target
        if train==None:
            self.train = self.dataset.train
        else:
            self.train = train
        self.learning_rate = learning_rate
        self.squashed = squashed
        self.input_features = [one]+dataset.input_features # one is defined below
        self.weights = {feat:random.uniform(-max_init,max_init)
                        for feat in self.input_features}
```

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

learnLinear.py — (continued)

```

    def predictor(self,e):
        """returns the prediction of the learner on example e"""
        linpred = sum(w*f(e) for f,w in self.weights.items())
        if self.squashed:
            return sigmoid(linpred)
        else:
            return linpred

    def predictor_string(self, sig_dig=3):
        """returns the doc string for the current prediction function
        sig_dig is the number of significant digits in the numbers"""
        doc = "+".join(str(round(val,sig_dig))+"*"+feat.__doc__
                        for feat,val in self.weights.items())
        if self.squashed:
            return "sigmoid("+ doc+")"
        else:
            return doc
```

learn is the main algorithm of the learner. It does *num_iter* steps of stochastic gradient descent with batch size = 1. The other parameters it gets from the class.

learnLinear.py — (continued)

```
    def learn(self,num_iter=100):
        for it in range(num_iter):
            self.display(2,"prediction=",self.predictor_string())
            for e in self.train:
                predicted = self.predictor(e)
                error = self.target(e) - predicted
                update = self.learning_rate*error
                for feat in self.weights:
                    self.weights[feat] += update*feat(e)
        #self.predictor.__doc__ = self.predictor_string()
        #return self.predictor
```

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

learnLinear.py — (continued)

```
def one(e):
    "1"
    return 1
```

$sigmoid(x)$ is the function

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

learnLinear.py — (continued)

```
def sigmoid(x):
    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)

```
from learnProblem import Data_set, Data_from_file, evaluation_criteria
import matplotlib.pyplot as plt

def test(**args):
    data = Data_from_file('data/SPECT.csv', target_index=0)
    # data = Data_from_file('data/mail_reading.csv', target_index=-1)
    # data = Data_from_file('data/carbool.csv', target_index=-1)
    learner = Linear_learner(data,**args)
    learner.learn()
    print("function learned is", learner.predictor_string())
    for ecrit in evaluation_criteria:
        test_error = data.evaluate_dataset(data.test, learner.predictor, ecrit)
        print("   Average", ecrit.__doc__, "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)

```
def plot_steps(learner=None,
               data = None,
               criterion="sum-of-squares",
               step=1,
               num_steps=1000,
               log_scale=True,
               label=""):
    """
    plots the training and test error for a learner.
    data is the
    learner_class is the class of the learning algorithm
    criterion gives the evaluation criterion plotted on the y-axis
    step specifies how many steps are run for each point on the plot
    num_steps is the number of points to plot

    """
    plt.ion()
    plt.xlabel("step")
    plt.ylabel("Average "+criterion+" error")
    if log_scale:
        plt.xscale('log') #plt.semilogx() #Makes a log scale
    else:
        plt.xscale('linear')
    if data is None:
        data = Data_from_file('data/holiday.csv', num_train=19, target_index=-1)
        #data = Data_from_file('data/SPECT.csv', target_index=0)
        # data = Data_from_file('data/mail_reading.csv', target_index=-1)
        # data = Data_from_file('data/carbool.csv', target_index=-1)
    random.seed(None)  # reset seed
    if learner is None:
        learner = Linear_learner(data)
    train_errors = []
    test_errors = []
    for i in range(1,num_steps+1,step):
        test_errors.append(data.evaluate_dataset(data.test, learner.predictor, criterion))
        train_errors.append(data.evaluate_dataset(data.train, learner.predictor, criterion))
        learner.display(2, "Train error:",train_errors[-1],
                        "Test error:",test_errors[-1])
        learner.learn(num_iter=step)
    plt.plot(range(1,num_steps+1,step),train_errors,ls='-',c='k',label="training errors")
    plt.plot(range(1,num_steps+1,step),test_errors,ls='--',c='k',label="test errors")
    plt.legend()
    plt.draw()
    learner.display(1, "Train error:",train_errors[-1],
                    "Test error:",test_errors[-1])

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

```
# This generates the figure
# from learnProblem import Data_set_augmented,prod_feat
# data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0)
# dataplus = Data_set_augmented(data,[],[prod_feat])
# plot_steps(data=data,num_steps=10000)
# plot_steps(data=dataplus,num_steps=10000) # warning very slow
```

Exercise 7.11 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)

```
def arange(start,stop,step):
    """returns enumeration of values in the range [start,stop) separated by step.
    like the built-in range(start,stop,step) but allows for integers and floats.
    Note that rounding errors are expected with real numbers. (or use numpy.arange)
    """
    while start<stop:
        yield start
        start += step

def plot_prediction(learner=None,
             data = None,
             minx = 0,
             maxx = 5,
             step_size = 0.01,  # for plotting
             label="function"):
    plt.ion()
    plt.xlabel("x")
    plt.ylabel("y")
    if data is None:
        data = Data_from_file('data/simp_regr.csv', prob_test=0,
                        boolean_features=False, target_index=-1)
    if learner is None:
        learner = Linear_learner(data,squashed=False)
    learner.learning_rate=0.001
    learner.learn(100)
    learner.learning_rate=0.0001
    learner.learn(1000)
    learner.learning_rate=0.00001
    learner.learn(10000)
    learner.display(1,"function learned is", learner.predictor_string(),
            "error=",data.evaluate_dataset(data.train, learner.predictor, "sum-of-squares"))
    plt.plot([e[0] for e in data.train],[e[-1] for e in data.train],"bo",label="data")
    plt.plot(list(arange(minx,maxx,step_size)),[learner.predictor([x])
                                    for x in arange(minx,maxx,step_size)],
```

```
                                    label=label)
    plt.legend()
    plt.draw()
```

learnLinear.py — (continued)

```
from learnProblem import Data_set_augmented, power_feat
def plot_polynomials(data=None,
               learner_class = Linear_learner,
               max_degree=5,
               minx = 0,
               maxx = 5,
               num_iter = 100000,
               learning_rate = 0.0001,
               step_size = 0.01, # for plotting
               ):
    plt.ion()
    plt.xlabel("x")
    plt.ylabel("y")
    if data is None:
        data = Data_from_file('data/simp_regr.csv', prob_test=0,
                              boolean_features=False, target_index=-1)
    plt.plot([e[0] for e in data.train],[e[-1] for e in data.train],"ko",label="data")
    x_values = list(arange(minx,maxx,step_size))
    line_styles = ['-','--','-.',':']
    colors = ['0.5','k','k','k','k']
    for degree in range(max_degree):
        data_aug = Data_set_augmented(data,[power_feat(n) for n in range(1,degree+1)],
                                      include_orig=False)
        learner = learner_class(data_aug,squashed=False)
        learner.learning_rate=learning_rate
        learner.learn(num_iter)
        learner.display(1,"For degree",degree,
                    "function learned is", learner.predictor_string(),
                    "error=",data.evaluate_dataset(data.train, learner.predictor, "sum-of-squares"))
        ls = line_styles[degree % len(line_styles)]
        col = colors[degree % len(colors)]
        plt.plot(x_values,[learner.predictor([x]) for x in x_values], linestyle=ls, color=col,
                    label="degree="+str(degree))
        plt.legend(loc='upper left')
        plt.draw()

# Try:
# plot_prediction()
# plot_polynomials()
#data = Data_from_file('data/mail_reading.csv', target_index=-1)
#plot_prediction(data=data)
```

7.6.1 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 the 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 me made more fair?)

learnLinearBSGD.py — Linear Learner with Batched Stochastic Gradient Descent

```
from learnLinear import Linear_learner
import random, math

class Linear_learner_bsgd(Linear_learner):
    def __init__(self, *args, batch_size=10, **kargs):
        Linear_learner.__init__(self, *args, **kargs)
        self.batch_size = batch_size

    def learn(self,num_iter=None):
        if num_iter is None:
            num_iter = self.number_iterations
        batch_size = min(self.batch_size, len(self.train))
        d = {feat:0 for feat in self.weights}
        for it in range(num_iter):
            self.display(2,"prediction=",self.predictor_string())
            for e in random.sample(self.train, batch_size):
                predicted = self.predictor(e)
                error = self.target(e) - predicted
                update = self.learning_rate*error
                for feat in self.weights:
                    d[feat] += update*feat(e)
            for feat in self.weights:
                self.weights[feat] += d[feat]
                d[feat]=0

# from learnLinear import plot_steps
# from learnProblem import Data_from_file
# data = Data_from_file('data/holiday.csv', target_index=-1)
# learner = Linear_learner_bsgd(data)
# plot_steps(learner = learner, data=data)

# to plot polynomials with batching (compare to SGD)
# from learnLinear import plot_polynomials
# plot_polynomials(learner_class = Linear_learner_bsgd)
```

7.7 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

```
from learnProblem import Learner, Data_set, Data_from_file
from learnLinear import sigmoid, one
import random, math

class Layer(object):
    def __init__(self,nn,num_outputs=None):
        """Given a list of inputs, outputs will produce a list of length num_outputs.
        nn is the neural network this is part of
        num outputs is the number of outputs for this layer.
        """
        self.nn = nn
        self.num_inputs = nn.num_outputs # output of nn is the input to this layer
        if num_outputs:
            self.num_outputs = num_outputs
        else:
            self.num_outputs = nn.num_outputs # same as the inputs

    def output_values(self,input_values):
        """Return the outputs for this layer for the given input values.
        input_values is a list of the inputs to this layer (of length num_inputs)
        returns a list of length self.num_outputs
        """
        raise NotImplementedError("output_values") # abstract method

    def backprop(self,errors):
        """Backpropagate the errors on the outputs, return the errors on the inputs.
        errors is a list of errors for the outputs (of length self.num_outputs).
        Return the errors for the inputs to this layer (of length self.num_inputs).
        You can assume that this is only called after corresponding output_values,
           and it can remember information information required for the back-propagation.
        """
        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)

```
class Linear_complete_layer(Layer):
    """a completely connected layer"""
    def __init__(self, nn, num_outputs, max_init=0.2):
        """A completely connected linear layer.
        nn is a neural network that the inputs come from
        num_outputs is the number of outputs
        max_init is the maximum value for random initialization of parameters
        """
        Layer.__init__(self, nn, num_outputs)
        # self.weights[o][i] is the weight between input i and output o
```

```
        self.weights = [[random.uniform(-max_init, max_init)
                          for inf in range(self.num_inputs+1)]
                        for outf in range(self.num_outputs)]

    def output_values(self,input_values):
        """Returns the outputs for the input values.
        It remembers the values for the backprop.

        Note in self.weights there is a weight list for every output,
        so wts in self.weights effectively loops over the outputs.
        """
        self.inputs = input_values + [1]
        return [sum(w*val for (w,val) in zip(wts,self.inputs))
                    for wts in self.weights]

    def backprop(self,errors):
        """Backpropagate the errors, updating the weights and returning the error in its inputs.
        """
        input_errors = [0]*(self.num_inputs+1)
        for out in range(self.num_outputs):
            for inp in range(self.num_inputs+1):
                input_errors[inp] += self.weights[out][inp] * errors[out]
                self.weights[out][inp] += self.nn.learning_rate * self.inputs[inp] * errors[out]
        return input_errors[:-1] # remove the error for the "1"
```

learnNN.py — (continued)

```
class Sigmoid_layer(Layer):
    """sigmoids of the inputs.
    The number of outputs is equal to the number of inputs.
    Each output is the sigmoid of its corresponding input.
    """
    def __init__(self, nn):
        Layer.__init__(self, nn)

    def output_values(self,input_values):
        """Returns the outputs for the input values.
        It remembers the output values for the backprop.
        """
        self.outputs= [sigmoid(inp) for inp in input_values]
        return self.outputs

    def backprop(self,errors):
        """Returns the derivative of the errors"""
        return [e*out*(1-out) for e,out in zip(errors, self.outputs)]
```

learnNN.py — (continued)

```
class ReLU_layer(Layer):
    """Rectified linear unit (ReLU) f(z) = max(0, z).
    The number of outputs is equal to the number of inputs.
    """
```

```
    def __init__(self, nn):
        Layer.__init__(self, nn)

    def output_values(self,input_values):
        """Returns the outputs for the input values.
        It remembers the input values for the backprop.
        """
        self.input_values = input_values
        self.outputs= [max(0,inp) for inp in input_values]
        return self.outputs

    def backprop(self,errors):
        """Returns the derivative of the errors"""
        return [e if inp>0 else 0 for e,inp in zip(errors, self.input_values)]
```

learnNN.py — (continued)

```
class NN(Learner):
    def __init__(self, dataset, learning_rate=0.1):
        self.dataset = dataset
        self.learning_rate = learning_rate
        self.input_features = dataset.input_features
        self.num_outputs = len(self.input_features)
        self.layers = []

    def add_layer(self,layer):
        """add a layer to the network.
        Each layer gets values from the previous layer.
        """
        self.layers.append(layer)
        self.num_outputs = layer.num_outputs

    def predictor(self,ex):
        """Predicts the value of the first output feature for example ex.
        """
        values = [f(ex) for f in self.input_features]
        for layer in self.layers:
            values = layer.output_values(values)
        return values[0]

    def predictor_string(self):
        return "not implemented"
```

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

learnNN.py — (continued)

```

    def learn(self,num_iter):
        """Learns parameters for a neural network using stochastic gradient decent.
        num_iter is the number of iterations
        """
        for i in range(num_iter):
```

```
            for e in random.sample(self.dataset.train,len(self.dataset.train)):
                # compute all outputs
                values = [f(e) for f in self.input_features]
                for layer in self.layers:
                    values = layer.output_values(values)
                # backpropagate
                errors = self.sum_squares_error([self.dataset.target(e)],values)
                for layer in reversed(self.layers):
                    errors = layer.backprop(errors)

    def sum_squares_error(self,observed,predicted):
        """Returns the errors for each of the target features.
        """
        return [obsd-pred for obsd,pred in zip(observed,predicted)]
```

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

learnNN.py — (continued)

```
data = Data_from_file('data/mail_reading.csv', target_index=-1)
#data = Data_from_file('data/mail_reading_consis.csv', target_index=-1)
#data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0)
#data = Data_from_file('data/holiday.csv', target_index=-1) #, num_train=19)
nn1 = NN(data)
nn1.add_layer(Linear_complete_layer(nn1,3))
nn1.add_layer(Sigmoid_layer(nn1)) # comment this or the next
# nn1.add_layer(ReLU_layer(nn1))
nn1.add_layer(Linear_complete_layer(nn1,1))
nn1.add_layer(Sigmoid_layer(nn1))
nn1.learning_rate=0.1
#nn1.learn(100)

from learnLinear import plot_steps
import time
start_time = time.perf_counter()
plot_steps(learner = nn1, data = data, num_steps=10000)
for eg in data.train:
    print(eg,nn1.predictor(eg))
end_time = time.perf_counter()
print("Time:", end_time - start_time)
```

Exercise 7.12 In the definition of *nn*1 above, for each of the following, first hypothesize what will happen, then test your hypothesis, then explain whether you testing confirms your hypothesis or not. Test it for more than one data set, and use more than one run for each data set.

(a) Which fits the data better, having a sigmoid layer or a ReLU layer after the first linear layer?

(b) Which is faster, having a sigmoid layer or a ReLU layer after the first linear layer?

(c) What happens if you have both the sigmoid layer and then a ReLU layer after the first linear layer and before the second linear layer?

(d) What happens if you have neither the sigmoid layer nor a ReLU layer after the first linear layer?

(e) What happens if you have a ReLU layer then a sigmoid layer after the first linear layer and before the second linear layer?

Exercise 7.13 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
```

7.8 Boosting

The following code implements functional gradient boosting for regression.

A Boosted dataset is created from a base dataset by subtracting the prediction of the offset function from each example. This does not save the new dataset, but generates it as needed. The amount of space used is constant, independent on the size of the data set.

learnBoosting.py — Functional Gradient Boosting

```
from learnProblem import Data_set, Learner

class Boosted_dataset(Data_set):
    def __init__(self, base_dataset, offset_fun):
        """new dataset which is like base_dataset,
          but offset_fun(e) is subtracted from the target of each example e
        """
        self.base_dataset = base_dataset
        self.offset_fun = offset_fun
        Data_set.__init__(self, base_dataset.train, base_dataset.test,
                          base_dataset.prob_test, base_dataset.target_index)
```

```

    def create_features(self):
        self.input_features = self.base_dataset.input_features
        def newout(e):
            return self.base_dataset.target(e) - self.offset_fun(e)
        newout.frange = self.base_dataset.target.frange
        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)

```
class Boosting_learner(Learner):
    def __init__(self, dataset, base_learner_class):
        self.dataset = dataset
        self.base_learner_class = base_learner_class
        mean = sum(self.dataset.target(e)
                   for e in self.dataset.train)/len(self.dataset.train)
        self.predictor = lambda e:mean  # function that returns mean for each example
        self.predictor.__doc__ = "lambda e:"+str(mean)
        self.offsets = [self.predictor]
        self.errors = [data.evaluate_dataset(data.test, self.predictor, "sum-of-squares")]
        self.display(1,"Predict mean test set error=", self.errors[0] )


    def learn(self, num_ensemble=10):
        """adds num_ensemble learners to the ensemble.
        returns a new predictor.
        """
        for i in range(num_ensemble):
            train_subset = Boosted_dataset(self.dataset, self.predictor)
            learner = self.base_learner_class(train_subset)
            new_offset = learner.learn()
            self.offsets.append(new_offset)
            def new_pred(e, old_pred=self.predictor, off=new_offset):
                return old_pred(e)+off(e)
            self.predictor = new_pred
            self.errors.append(data.evaluate_dataset(data.test, self.predictor,"sum-of-squares"))
            self.display(1,"After Iteration",len(self.offsets)-1,"test set error=", self.errors[-1]
        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)

```
# Testing

from learnDT import DT_learner
```

```
from learnProblem import Data_set, Data_from_file

def sp_DT_learner(min_prop=0.9):
    def make_learner(dataset):
        mne = len(dataset.train)*min_prop
        return DT_learner(dataset,min_number_examples=mne)
    return make_learner

data = Data_from_file('data/carbool.csv', target_index=-1)
#data = Data_from_file('data/SPECT.csv', target_index=0)
#data = Data_from_file('data/mail_reading.csv', target_index=-1)
#data = Data_from_file('data/holiday.csv', num_train=19, target_index=-1)
learner9 = Boosting_learner(data, sp_DT_learner(0.9))
#learner7 = Boosting_learner(data, sp_DT_learner(0.7))
#learner5 = Boosting_learner(data, sp_DT_learner(0.5))
predictor9 =learner9.learn(10)
for i in learner9.offsets: print(i.__doc__)
import matplotlib.pyplot as plt

def plot_boosting(data,steps=10, thresholds=[0.5,0.1,0.01,0.001], markers=['-','--','-.',':'] ):
    learners = [Boosting_learner(data, sp_DT_learner(th)) for th in thresholds]
    predictors = [learner.learn(steps) for learner in learners]
    plt.ion()
    plt.xscale('linear') # change between log and linear scale
    plt.xlabel("number of trees")
    plt.ylabel(" error")
    for (learner,(threshold,marker)) in zip(learners,zip(thresholds,markers)):
        plt.plot(range(len(learner.errors)), learner.errors, ls=marker,c='k',
                 label=str(round(threshold*100))+"% min example threshold")
    plt.legend()
    plt.draw()

# plot_boosting(data)
```

Chapter 8

Reasoning Under Uncertainty

8.1 Representing Probabilistic Models

In the implementation of probabilistic models we will assume that 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.

probVariables.py — Probabilistic Variables

```python
import random

class Variable(object):
    """A random variable.
    name (string) - name of the variable
    domain (list) - a list of the values for the variable.
    Variables are ordered according to their name.
    """

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

```

    def __str__(self):
        return self.name

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

8.2 Representing Factors

A **factor** is, mathematically, a function from variables into a number; that is given a value for each of its variable, it gives a number. Factors are used for conditional probabilities, utilities in the next chapter, and are explictly constructed by some algorithms (in particular variable elimination).

A variable assignment, or just **assignment**, is represented as a $\{variable : value\}$ dictionary. A factor can be evaluated when all of its variables are assigned. The method get_value evaluates the factor for an assignment. The assignment can include extra variables not in the factor. This method needs to be defined for every subclass.

probFactors.py — Factors for graphical models

```
from display import Displayable
import math

class Factor(Displayable):
    nextid=0 # each factor has a unique identifier; for printing

    def __init__(self,variables):
        self.variables = variables # ordered list of variables
        self.id = Factor.nextid
        self.name = f"f{self.id}"
        Factor.nextid += 1

    def can_evaluate(self,assignment):
        """True when the factor can be evaluated in the assignment
        assignment is a {variable:value} dict
        """
        return all(v in assignment for v in self.variables)

    def get_value(self,assignment):
        """Returns the value of the factor given the assignment of values to variables.
        Needs to be defined for each subclass.
        """
        assert self.can_evaluate(assignment)
        raise NotImplementedError("get_value") # abstract method
```

The method __str__ returns a brief definition (like "f7(X,Y,Z)").The method to_table returns string representations of a table showing all of the assignments of values to variables, and the corresponding value.

probFactors.py — (continued)

```
    def __str__(self):
        """returns a string representing a summary of the factor"""
        return f"{self.name}({','.join(str(var) for var in self.variables)})"

    def to_table(self, variables=None):
        """returns a string representation of the factor.
        Allows for an arbitrary variable ordering.
        variables is a list of the variables in the factor
        (can contain other variables)"""
        if variables==None:
            variables = self.variables
        else:  #enforce ordering and allow for extra variables in ordering
            variables = [v for v in variables if v in self.variables]
        head = "\t".join(str(v) for v in variables)
        return head+"\n"+self.ass_to_str(variables, {}, variables)

    def ass_to_str(self, vars, asst, allvars):
        #print(f"ass_to_str({vars}, {asst}, {allvars})")
        if vars:
            return "\n".join(self.ass_to_str(vars[1:], asst | {vars[0]:val}, allvars)
                            for val in vars[0].domain)
        else:
            return ("\t".join(str(asst[var]) for var in allvars)
                        + "\t"+str(self.get_value(asst)) )

    __repr__ = __str__
```

8.3 Conditional Probability Distributions

A **conditional probability distribution (CPD)** is a type of factor that represents a conditional probability. A CPD representing $P(X \mid Y_1 \dots Y_k)$ is a type of factor, where given values for X and each Y_i returns a number.

probFactors.py — (continued)

```
class CPD(Factor):
    def __init__(self, child, parents):
        """represents P(variable | parents)
        """
        self.parents = parents
        self.child = child
        Factor.__init__(self, parents+[child])

    def __str__(self):
        """A brief description of a factor using in tracing"""
        if self.parents:
            return f"P({self.child}|{','.join(str(p) for p in self.parents)})"
        else:
            return f"P({self.child})"
```

```

    __repr__ = __str__
```

8.3.1 Logistic Regression

A **logistic regression** CPD, for Boolean variable X represents $P(X{=}True \mid Y_1 \ldots Y_k)$, using $k+1$ real-values weights so

$$P(X{=}True \mid Y_1 \ldots Y_k) = sigmoid(w_0 + \sum_i w_i Y_i)$$

where for Boolean Y_i, True is represented as 1 and False as 0.

probFactors.py — (continued)

```
from learnLinear import sigmoid

class LogisticRegression(CPD):
    def __init__(self, child, parents, weights):
        """A logistic regression representation of a conditional probability.
        child is the Boolean (or 0/1) variable whose CPD is being defined
        parents is the list of parents
        weights is list of parameters, such that weights[i+1] is the weight for parents[i]
        """
        assert len(weights) == 1+len(parents)
        CPD.__init__(self, child, parents)
        self.weights = weights

    def get_value(self,assignment):
        assert self.can_evaluate(assignment)
        prob = sigmoid(self.weights[0]
                        + sum(self.weights[i+1]*assignment[self.parents[i]]
                                  for i in range(len(self.parents))))
        if assignment[self.child]: #child is true
            return prob
        else:
            return (1-prob)
```

8.3.2 Noisy-or

A **noisy-or**, for Boolean variable X with Boolean parents $Y_1 \ldots Y_k$ is parametrized by $k+1$ parameters $p_0, p_1, \ldots, p_k$, where each $0 \leq p_i \leq 1$. The sematics is defined as though there are $k+1$ hidden variables $Z_0, Z_1 \ldots Z_k$, where $P(Z_0) = p_0$ and $P(Z_i \mid Y_i) = p_i$ for $i \geq 1$, and where X is true if and only if $Z_0 \vee Z_1 \vee \cdots \vee Z_k$ (where $\vee$ is "or"). Thus X is false if all of the Z_i are false. Intuitively, Z_0 is the probability of X when all Y_i are false and each Z_i is a noisy (probabilistic) measure that Y_i makes X true, and X only needs one to make it true.

probFactors.py — (continued)

```
class NoisyOR(CPD):
    def __init__(self, child, parents, weights):
        """A noisy representation of a conditional probability.
        variable is the Boolean (or 0/1) child variable whose CPD is being defined
        parents is the list of Boolean (or 0/1) parents
        weights is list of parameters, such that weights[i+1] is the weight for parents[i]
        """
        assert len(weights) == 1+len(parents)
        CPD.__init__(self, child, parents)
        self.weights = weights

    def get_value(self,assignment):
        assert self.can_evaluate(assignment)
        probfalse = (1-self.weights[0])*math.prod(1-self.weights[i+1]
                                                  for i in range(len(self.parents))
                                                  if assignment[self.parents[i]])
        if assignment[self.child]:
            return 1-probfalse
        else:
            return probfalse
```

8.3.3 Tabular Factors

A **tabular factor** is a factor that represents each assignment of values to variables separately. It is represented by a Python array (or python dict). If the variables are $V_1, V_2, \dots, V_k$, the value of $f(V_1 = v_1, V_2 = v_1, \dots, V_k = v_k)$ is stored in $f[v_1][v_2] \dots [v_k]$.

If the domain of V_i is $[0, \dots, n_i - 1]$ this can be represented as an array. Otherwise we can use a dictionary. Python is nice in that it doesn't care, whether an array or dict is used **except when enumerating the values**; enumerating a dict gives the keys (the variables) but enumerating an array gives the values. So we have to be careful not to do this.

probFactors.py — (continued)

```
from functools import reduce

class TabFactor(Factor):

    def __init__(self, variables, values):
        Factor.__init__(self, variables)
        self.values = values

    def get_value(self, assignment):
        return self.get_val_rec(self.values, self.variables, assignment)

    def get_val_rec(self, value, variables, assignment):
        if variables == []:
           return value
        else:
```

```
        return self.get_val_rec(value[assignment[variables[0]]],
                                variables[1:],assignment)
```

Prob is a factor that represents a conditional probability by enumerating all of the values.

probFactors.py — (continued)

```
class Prob(CPD,TabFactor):
    """A factor defined by a conditional probability table"""
    def __init__(self,var,pars,cpt):
        """Creates a factor from a conditional probability table, cpt
        The cpt values are assumed to be for the ordering par+[var]
        """
        TabFactor.__init__(self,pars+[var],cpt)
        self.child = var
        self.parents = pars
```

8.4 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

```
from display import Displayable
from probFactors import CPD
import matplotlib.pyplot as plt

class GraphicalModel(Displayable):
    """The class of graphical models.
    A graphical model consists of a title, a set of variables and a set of factors.

    vars is a set of variables
    factors is a set of factors
    """
    def __init__(self, title, variables=None, factors=None):
        self.title = title
        self.variables = variables
        self.factors = factors
```

A **belief network** (also known as a **Bayesian network**) is a graphical model where all of the factors are conditional probabilities, and every variable has a conditional probability of it given its parents. This only checks the first condition, and builds some useful data structures.

probGraphicalModels.py — (continued)

```
class BeliefNetwork(GraphicalModel):
```

```
    """The class of belief networks."""

    def __init__(self, title, variables, factors):
        """vars is a set of variables
        factors is a set( of factors. All of the factors are instances of CPD (e.g., Prob).
        """
        GraphicalModel.__init__(self, title, variables, factors)
        assert all(isinstance(f,CPD) for f in factors)
        self.var2cpt = {f.child:f for f in factors}
        self.var2parents = {f.child:f.parents for f in factors}
        self.children = {n:[] for n in self.variables}
        for v in self.var2parents:
            for par in self.var2parents[v]:
                self.children[par].append(v)
        self.topological_sort_saved = None
```

The following creates a topological sort of the nodes, where the parents of a node come before the node in the resulting order. This is based on Kahn's algorithm from 1962.

probGraphicalModels.py — (continued)

```
    def topological_sort(self):
        """creates a topological ordering of variables such that the parents of
        a node are before the node.
        """
        if self.topological_sort_saved:
            return self.topological_sort_saved
        next_vars = {n for n in self.var2parents if not self.var2parents[n] }
        self.display(3,'topological_sort: next_vars',next_vars)
        top_order=[]
        while next_vars:
            var = next_vars.pop()
            self.display(3,'select variable',var)
            top_order.append(var)
            next_vars |= {ch for ch in self.children[var]
                             if all(p in top_order for p in self.var2parents[ch])}
            self.display(3,'var_with_no_parents_left',next_vars)
        self.display(3,"top_order",top_order)
        assert set(top_order)==set(self.var2parents),(top_order,self.var2parents)
        self.topologicalsort_saved=top_order
        return top_order
```

The **show** method uses matplotlib to show the graphical structure of a belief network.

probGraphicalModels.py — (continued)

```
    def show(self):
        plt.ion()  # interactive
        ax = plt.figure().gca()
        ax.set_axis_off()
        plt.title(self.title)
```

```
        bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5")
        for var in reversed(self.topological_sort()):
            if self.var2parents[var]:
                for par in self.var2parents[var]:
                    ax.annotate(var.name, par.position, xytext=var.position,
                                    arrowprops={'arrowstyle':'<-'},bbox=bbox,
                                    ha='center')
            else:
                x,y = var.position
                plt.text(x,y,var.name,bbox=bbox,ha='center')
```

8.4.1 Example Belief Networks

A Chain of 4 Variables

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

Please do not change this, as it is the example used for testing.

probGraphicalModels.py — (continued)

```
from probVariables import Variable
from probFactors import Prob, LogisticRegression, NoisyOR

boolean = [False, True]
A = Variable("A", boolean, position=(0,0.8))
B = Variable("B", boolean, position=(0.333,0.6))
C = Variable("C", boolean, position=(0.666,0.4))
D = Variable("D", boolean, position=(1,0.2))

f_a = Prob(A,[],[0.4,0.6])
f_b = Prob(B,[A],[[0.9,0.1],[0.2,0.8]])
f_c = Prob(C,[B],[[0.6,0.4],[0.3,0.7]])
f_d = Prob(D,[C],[[0.1,0.9],[0.75,0.25]])

bn_4ch = BeliefNetwork("4-chain", {A,B,C,D}, {f_a,f_b,f_c,f_d})
```

Report-of-Leaving Example

The second beleif network, `bn_report`, is Example 8.15 of Poole and Mackworth [2017] (`http://artint.info`). The output of `bn_report.show()` is shown in Figure 8.1 of this document.

probGraphicalModels.py — (continued)

```
# Beleif network report-of-leaving example (Example 8.15 shown in Figure 8.3) of
# Poole and Mackworth, Artificial Intelligence, 2017 http://artint.info

Alarm =  Variable("Alarm", boolean, position=(0.366,0.633))
Fire =   Variable("Fire",  boolean, position=(0.633,0.9))
Leaving = Variable("Leaving", boolean, position=(0.366,0.366))
Report = Variable("Report", boolean, position=(0.366,0.1))
```

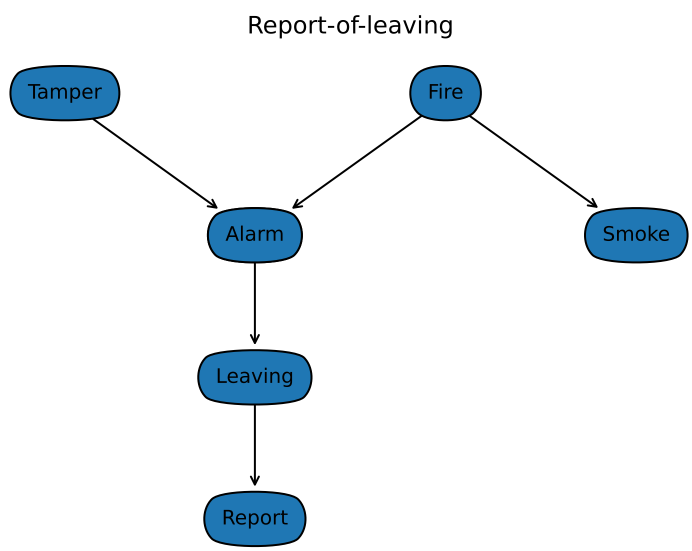


Figure 8.1: The report-of-leaving belief network

```
Smoke =  Variable("Smoke", boolean, position=(0.9,0.633))
Tamper = Variable("Tamper", boolean, position=(0.1,0.9))

f_ta = Prob(Tamper,[],[0.98,0.02])
f_fi = Prob(Fire,[],[0.99,0.01])
f_sm = Prob(Smoke,[Fire],[[0.99,0.01],[0.1,0.9]])
f_al = Prob(Alarm,[Fire,Tamper],[[[0.9999, 0.0001], [0.15, 0.85]], [[0.01, 0.99], [0.5, 0.5]]])
f_lv = Prob(Leaving,[Alarm],[[0.999, 0.001], [0.12, 0.88]])
f_re = Prob(Report,[Leaving],[[0.99, 0.01], [0.25, 0.75]])

bn_report = BeliefNetwork("Report-of-leaving", {Tamper,Fire,Smoke,Alarm,Leaving,Report},
                           {f_ta,f_fi,f_sm,f_al,f_lv,f_re})
```

Sprinkler Example

The third belief network is the sprinkler example from Pearl. The output of `bn_sprinkler.show()` is shown in Figure 8.2 of this document.

probGraphicalModels.py — (continued)

```
Season = Variable("Season", ["summer","winter"], position=(0.5,0.9))
Sprinkler = Variable("Sprinkler", ["on","off"], position=(0.9,0.6))
Rained = Variable("Rained", boolean, position=(0.1,0.6))
Grass_wet = Variable("Grass wet", boolean, position=(0.5,0.3))
Grass_shiny = Variable("Grass shiny", boolean, position=(0.1,0))
```

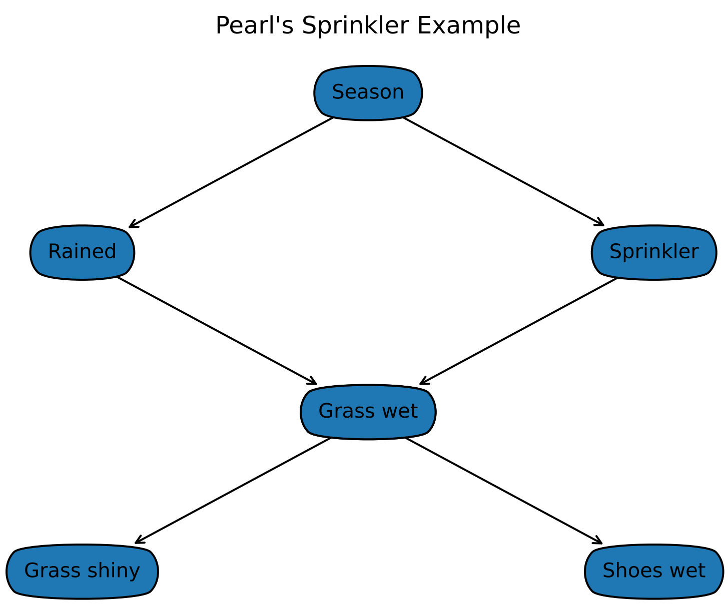


Figure 8.2: The sprinkler belief network

```
Shoes_wet = Variable("Shoes wet", boolean, position=(0.9,0))

f_season = Prob(Season,[],[0.5,0.5])
f_sprinkler = Prob(Sprinkler,[Season],[0.9,0.1,0.05,0.95])
f_rained = Prob(Rained,[Season],[0.7,0.3,0.2,0.8])
f_wet = Prob(Grass_wet,[Sprinkler,Rained], [1,0,0.1,0.9,0.2,0.8,0.02,0.98])
f_shiny = Prob(Grass_shiny, [Grass_wet], [0.95,0.05,0.3,0.7])
f_shoes = Prob(Shoes_wet, [Grass_wet], [0.92,0.08,0.35,0.65])

bn_sprinkler = BeliefNetwork("Pearl's Sprinkler Example",
                        {Season, Sprinkler, Rained, Grass_wet, Grass_shiny, Shoes_wet},
                        {f_season, f_sprinkler, f_rained, f_wet, f_shiny, f_shoes})
```

Bipartite diagnostic model with noisy-or

This belief network is a bipartite diagnostic model, with independent diseases, and the symtoms depend on the diseases, where the CPDs are defined using noisy-or. Bipartite means it is in two parts; the diseases are only connected to the symptomes and the symptoms are only connected to the diseases. The output of `bn_no1.show()` is shown in Figure 8.3 of this document.

probGraphicalModels.py — (continued)

```
Cough = Variable("Cough", boolean, (0.1,0.1))
Fever = Variable("Fever", boolean, (0.5,0.1))
```

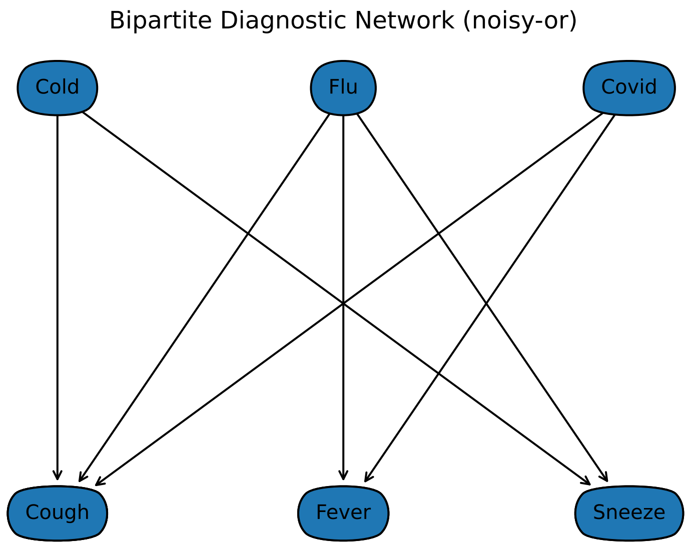


Figure 8.3: A partite diagnostic network

```
Sneeze = Variable("Sneeze", boolean, (0.9,0.1))
Cold = Variable("Cold",boolean, (0.1,0.9))
Flu = Variable("Flu",boolean, (0.5,0.9))
Covid = Variable("Covid",boolean, (0.9,0.9))

p_cold_no = Prob(Cold,[],[0.9,0.1])
p_flu_no = Prob(Flu,[],[0.95,0.05])
p_covid_no = Prob(Covid,[],[0.99,0.01])

p_cough_no = NoisyOR(Cough, [Cold,Flu,Covid], [0.1, 0.3, 0.2, 0.7])
p_fever_no = NoisyOR(Fever, [    Flu,Covid], [0.01,     0.6,  0.7])
p_sneeze_no = NoisyOR(Sneeze, [Cold,Flu  ], [0.05, 0.5,  0.2    ])

bn_no1 = BeliefNetwork("Bipartite Diagnostic Network (noisy-or)",
                      {Cough, Fever, Sneeze, Cold, Flu, Covid},
                       {p_cold_no, p_flu_no, p_covid_no, p_cough_no, p_fever_no, p_sneeze_no})

# to see the conditional probability of Noisy-or do:
#print(p_cough_no.to_table())
```

Bipartite diagnostic model with noisy-or

The next belief network is a bipartite diagnostic model, with independent diseases, and the symtoms depend on the diseases, where the CPDs are defined using logistic regression. It has the same graphical structure as the previous example (see Figure 8.3). This has the (approximately) the same conditional probabilities as the previous example when zero or one diseases are present. Note that $sigmoid(-2.2) \approx 0.1$

probGraphicalModels.py — (continued)

```

p_cold_lr = Prob(Cold,[],[0.9,0.1])
p_flu_lr = Prob(Flu,[],[0.95,0.05])
p_covid_lr = Prob(Covid,[],[0.99,0.01])

p_cough_lr = LogisticRegression(Cough, [Cold,Flu,Covid], [-2.2, 1.67, 1.26, 3.19])
p_fever_lr = LogisticRegression(Fever, [  Flu,Covid], [-4.6,      5.02, 5.46])
p_sneeze_lr = LogisticRegression(Sneeze, [Cold,Flu ], [-2.94, 3.04, 1.79  ])

bn_lr1 = BeliefNetwork("Bipartite Diagnostic Network - logistic regression",
                       {Cough, Fever, Sneeze, Cold, Flu, Covid},
                        {p_cold_lr, p_flu_lr, p_covid_lr, p_cough_lr, p_fever_lr, p_sneeze_lr})

# to see the conditional probability of Noisy-or do:
#print(p_cough_lr.to_table())
```

8.5 Inference Methods

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

probGraphicalModels.py — (continued)

```
from display import Displayable

class InferenceMethod(Displayable):
    """The abstract class of graphical model inference methods"""
    method_name = "unnamed" # each method should have a method name

    def query(self, qvar, obs={}):
        """returns a {value:prob} dictionary for the query variable"""
        raise NotImplementedError("InferenceMethod query") # abstract method
```

We use bn_4ch as the test case, in particular $P(B \mid D = true)$. This needs an error threshold, particularly for the approximate methods.

probGraphicalModels.py — (continued)

```
    def testIM(self, threshold=0.0000000001):
        solver = self(bn_4ch)
        res = solver.query(B,{D:True})
        correct_answer = 0.429632380245
```

```
        assert correct_answer-threshold < res[True] < correct_answer+threshold, f"value {res[True]}
        print(f"Unit test passed for {self.method_name}.")
```

8.6 Recursive Conditioning

An instance of a *RC* object takes in a graphical model. The query method uses recursive conditioning to compute the probability of a query variable given observations on other variables.

probRC.py — Recursive Conditioning for Graphical Models

```
import math
from probGraphicalModels import GraphicalModel, InferenceMethod
from probFactors import Factor
from utilities import dict_union

class RC(InferenceMethod):
    """The class that queries graphical models using recursive conditioning

    gm is graphical model to query
    """
    method_name = "recursive conditioning"

    def __init__(self,gm=None):
        self.gm = gm
        self.cache = {(frozenset(), frozenset()):1}
        ## self.max_display_level = 3

    def query(self,var,obs={},split_order=None):
        """computes P(var|obs) where
        var is a variable
        obs is a variable:value dictionary
        split_order is a list of the non-observed non-query variables in gm
        """
        if var in obs:
            return {val:(1 if val == obs[var] else 0) for val in var.domain}
        else:
           if split_order == None:
                split_order = [v for v in self.gm.variables if (v not in obs) and v != var]
           unnorm = [self.rc(dict_union({var:val},obs), self.gm.factors, split_order)
                         for val in var.domain]
           p_obs = sum(unnorm)
           return {val:pr/p_obs for val,pr in zip(var.domain, unnorm)}
```

The following us the simplest search-based algorithm. It is exponential in the number of variables, so is not very useful. However, it is simple, and useful to understand before looking at the more complicated algorithm. Note that the above code does not call `rc0`; you will need to change the `self.rc` to `self.rc0` in above code to use it.

probRC.py — (continued)

```
    def rc0(self, context, factors, split_order):
        """simplest search algorithm"""
        self.display(2,"calling rc0,",(context,factors))
        if not factors:
            return 1
        elif to_eval := {fac for fac in factors if fac.can_evaluate(context)}:
            self.display(3,"rc0 evaluating factors",to_eval)
            val = math.prod(fac.get_value(context) for fac in to_eval)
            return val * self.rc0(context, factors-to_eval, split_order)
        else:
            total = 0
            var = split_order[0]
            self.display(3, "rc0 branching on", var)
            for val in var.domain:
                total += self.rc0(dict_union({var:val},context), factors, split_order[1:])
            self.display(3, "rc0 branching on", var,"returning", total)
            return total
```

probRC.py — (continued)

```
    def rc(self, context, factors, split_order):
        """ returns the number \sum_{split_order} \prod_{factors} given assignments in context
        context is a variable:value dictionary
        factors is a set of factors
        split_order is a list of variables in factors that are not in context
        """
        self.display(3,"calling rc,",(context,factors))
        ce = (frozenset(context.items()), frozenset(factors)) # key for the cache entry
        if ce in self.cache:
            self.display(2,"rc cache lookup",(context,factors))
            return self.cache[ce]
#        if not factors: # no factors; needed if you don't have forgetting and caching
#            return 1
        elif vars_not_in_factors := {var for var in context
                                     if not any(var in fac.variables for fac in factors)}:
             # forget variables not in any factor
            self.display(3,"rc forgetting variables", vars_not_in_factors)
            return self.rc({key:val for (key,val) in context.items()
                                if key not in vars_not_in_factors},
                            factors, split_order)
        elif to_eval := {fac for fac in factors if fac.can_evaluate(context)}:
            # evaluate factors when all variables are assigned
            self.display(3,"rc evaluating factors",to_eval)
            val = math.prod(fac.get_value(context) for fac in to_eval)
            if val == 0:
                return 0
            else:
             return val * self.rc(context, {fac for fac in factors if fac not in to_eval}, split_order)
        elif len(comp := connected_components(context, factors, split_order)) > 1:
            # there are disconnected components
```

```
            self.display(2,"splitting into connected components",comp)
            return(math.prod(self.rc(context,f,eo) for (f,eo) in comp))
        else:
            assert split_order, "split_order should not be empty to get here"
            total = 0
            var = split_order[0]
            self.display(3, "rc branching on", var)
            for val in var.domain:
                total += self.rc(dict_union({var:val},context), factors, split_order[1:])
            self.cache[ce] = total
            self.display(2, "rc branching on", var,"returning", total)
            return total
```

`connected_components` returns a list of connected components, where a connected component is a set of factors and a set of variables, where the graph that connects variables and factors that involve them is connected. The connected components are built one at a time; with a current connected component. At all times `factors` is partitioned into 3 disjoint sets:

- `component_factors` containing factors in the current connected component where all factors that share a variable are already in the component
- `factors_to_check` containing factors in the current connected component where potentially some factors that share a variable are not in the component; these need to be checked
- `other_factors` the other factors that are not (yet) in the connected component

probRC.py — (continued)

```
def connected_components(context, factors, split_order):
    """returns a list of (f,e) where f is a subset of factors and e is a subset of split_order
    such that each element shares the same variables that are disjoint from other elements.
    """
    other_factors = set(factors) #copies factors
    factors_to_check = {other_factors.pop()} # factors in connected component still to be checked
    component_factors = set() # factors in first connected component already checked
    component_variables = set() # variables in first connected component
    while factors_to_check:
        next_fac = factors_to_check.pop()
        component_factors.add(next_fac)
        new_vars = set(next_fac.variables) - component_variables - context.keys()
        component_variables |= new_vars
        for var in new_vars:
            factors_to_check |= {f for f in other_factors if var in f.variables}
            other_factors -= factors_to_check # set difference
    if other_factors:
        return ( [(component_factors,[e for e in split_order if e in component_variables])]
                    + connected_components(context, other_factors, [e for e in split_order if e no
    else:
```

```
        return [(component_factors, split_order)]
```

Testing:

probRC.py — (continued)

```
from probGraphicalModels import bn_4ch, A,B,C,D,f_a,f_b,f_c,f_d
bn_4chv = RC(bn_4ch)
## bn_4chv.query(A,{})
## bn_4chv.query(D,{})
## InferenceMethod.max_display_level = 3 # show more detail in displaying
## InferenceMethod.max_display_level = 1 # show less detail in displaying
## bn_4chv.query(A,{D:True},[C,B])
## bn_4chv.query(B,{A:True,D:False})

from probGraphicalModels import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
bn_reportRC = RC(bn_report) # answers queries using recursive conditioning
## bn_reportRC.query(Tamper,{})
## InferenceMethod.max_display_level = 0 # show no detail in displaying
## bn_reportRC.query(Leaving,{})
## bn_reportvRC.query(Tamper,{}, split_order=[Smoke,Fire,Alarm,Leaving,Report])
## bn_reportRC.query(Tamper,{Report:True})
## bn_reportvRC.query(Tamper,{Report:True,Smoke:False})

from probGraphicalModels import bn_sprinkler, Season, Sprinkler, Rained, Grass_wet, Grass_shiny, Shoe
bn_sprinklerv = RC(bn_sprinkler)
## bn_sprinklerv.query(Shoes_wet,{})
## bn_sprinklerv.query(Shoes_wet,{Rained:True})
## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:True})
## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:False,Rained:True})

from probGraphicalModels import bn_no1, bn_lr1, Cough, Fever, Sneeze, Cold, Flu, Covid
bn_no1v = RC(bn_no1)
bn_lr1v = RC(bn_lr1)
## bn_lr1v.query(Flu, {Fever:1, Sneeze:1})
## bn_lr1v.query(Cough,{})
## bn_lr1v.query(Cold,{Cough:1,Sneeze:0,Fever:1})
## bn_lr1v.query(Flu,{Cough:0,Sneeze:1,Fever:1})
## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1})
## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:0})
## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:1})

if __name__ == "__main__":
    InferenceMethod.testIM(RC)
```

8.7 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

```
from probFactors import Factor, FactorObserved, FactorSum, factor_times
from probGraphicalModels import GraphicalModel, InferenceMethod

class VE(InferenceMethod):
    """The class that queries Graphical Models using variable elimination.

    gm is graphical model to query
    """
    method_name = "variable elimination"

    def __init__(self,gm=None):
        self.gm = gm

    def query(self,var,obs={},elim_order=None):
        """computes P(var|obs) where
        var is a variable
        obs is a {variable:value} dictionary"""
        if var in obs:
            return {var:1 if val == obs[var] else 0 for val in var.domain}
        else:
            if elim_order == None:
                elim_order = self.gm.variables
            projFactors = [self.project_observations(fact,obs)
                           for fact in self.gm.factors]
            for v in elim_order:
                if v != var and v not in obs:
                    projFactors = self.eliminate_var(projFactors,v)
            unnorm = factor_times(var,projFactors)
            p_obs=sum(unnorm)
            self.display(1,"Unnormalized probs:",unnorm,"Prob obs:",p_obs)
            return {val:pr/p_obs for val,pr in zip(var.domain, unnorm)}
```

A *FactorObserved* is a factor that is the result of some observations on another factor. We don't store the values in a list; we just look them up as needed. The observations can include variables that are not in the list, but should have some intersection with the variables in the factor.

probFactors.py — (continued)

```
class FactorObserved(Factor):
    def __init__(self,factor,obs):
        Factor.__init__(self, [v for v in factor.variables if v not in obs])
        self.observed = obs
        self.orig_factor = factor

    def get_value(self,assignment):
        ass = assignment.copy()
        for ob in self.observed:
            ass[ob]=self.observed[ob]
        return self.orig_factor.get_value(ass)
```

A *FactorSum* is a factor that is the result of summing out a variable from the product of other factors. I.e., it constructs a representation of:

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

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)

```
class FactorSum(Factor):
    def __init__(self,var,factors):
        self.var_summed_out = var
        self.factors = factors
        vars = []
        for fac in factors:
            for v in fac.variables:
                if v is not var and v not in vars:
                    vars.append(v)
        Factor.__init__(self,vars)
        self.values = {}

    def get_value(self,assignment):
        """lazy implementation: if not saved, compute it. Return saved value"""
        asst = frozenset(assignment.items())
        if asst in self.values:
            return self.values[asst]
        else:
            total = 0
            new_asst = assignment.copy()
            for val in self.var_summed_out.domain:
                new_asst[self.var_summed_out] = val
                total += math.prod(fac.get_value(new_asst) for fac in self.factors)
            self.values[asst] = total
            return total
```

The method *factor_times* multiples a set of factors that are all factors on the same variable (or on no variables). This is the last step in variable elimination before normalizing. It returns an array giving the product for each value of *variable*.

probFactors.py — (continued)

```
def factor_times(variable, factors):
    """when factors are factors just on variable (or on no variables)"""
    prods = []
    facs = [f for f in factors if variable in f.variables]
    for val in variable.domain:
        ast = {variable:val}
        prods.append(math.prod(f.get_value(ast) for f in facs))
    return prods
```

To project observations onto a factor, for each variable that is observed in the factor, we construct a new factor that is the factor projected onto that variable. *Factor_observed* creates a new factor that is the result is assigning a value to a single variable.

probVE.py — (continued)

```
    def project_observations(self,factor,obs):
        """Returns the resulting factor after observing obs

        obs is a dictionary of {variable:value} pairs.
        """
        if any((var in obs) for var in factor.variables):
            # a variable in factor is observed
            return FactorObserved(factor,obs)
        else:
            return factor

    def eliminate_var(self,factors,var):
        """Eliminate a variable var from a list of factors.
        Returns a new set of factors that has var summed out.
        """
        self.display(2,"eliminating ",str(var))
        contains_var = []
        not_contains_var = []
        for fac in factors:
            if var in fac.variables:
                contains_var.append(fac)
            else:
                not_contains_var.append(fac)
        if contains_var == []:
            return factors
        else:
            newFactor = FactorSum(var,contains_var)
            self.display(2,"Multiplying:",[str(f) for f in contains_var])
            self.display(2,"Creating factor:", newFactor)
            self.display(3, newFactor.to_table()) # factor in detail
            not_contains_var.append(newFactor)
            return not_contains_var

from probGraphicalModels import bn_4ch, A,B,C,D
bn_4chv = VE(bn_4ch)
## bn_4chv.query(A,{})
## bn_4chv.query(D,{})
## InferenceMethod.max_display_level = 3 # show more detail in displaying
## InferenceMethod.max_display_level = 1 # show less detail in displaying
## bn_4chv.query(A,{D:True})
## bn_4chv.query(B,{A:True,D:False})

from probGraphicalModels import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
bn_reportv = VE(bn_report) # answers queries using variable elimination
```

```
## bn_reportv.query(Tamper,{})
## InferenceMethod.max_display_level = 0 # show no detail in displaying
## bn_reportv.query(Leaving,{})
## bn_reportv.query(Tamper,{},elim_order=[Smoke,Report,Leaving,Alarm,Fire])
## bn_reportv.query(Tamper,{Report:True})
## bn_reportv.query(Tamper,{Report:True,Smoke:False})

from probGraphicalModels import bn_sprinkler, Season, Sprinkler, Rained, Grass_wet, Grass_shiny, Shoe
bn_sprinklerv = VE(bn_sprinkler)
## bn_sprinklerv.query(Shoes_wet,{})
## bn_sprinklerv.query(Shoes_wet,{Rained:True})
## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:True})
## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:False,Rained:True})

from probGraphicalModels import bn_lr1, Cough, Fever, Sneeze, Cold, Flu, Covid
vediag = VE(bn_lr1)
## vediag.query(Cough,{})
## vediag.query(Cold,{Cough:1,Sneeze:0,Fever:1})
## vediag.query(Flu,{Cough:0,Sneeze:1,Fever:1})
## vediag.query(Covid,{Cough:1,Sneeze:0,Fever:1})
## vediag.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:0})
## vediag.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:1})

if __name__ == "__main__":
    InferenceMethod.testIM(VE)
```

8.8 Stochastic Simulation

8.8.1 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

```
import random
from probGraphicalModels import InferenceMethod

def sample_one(dist):
    """returns the index of a single sample from normalized distribution dist."""
    rand = random.random()*sum(dist.values())
    cum = 0    # cumulative weights
    for v in dist:
        cum += dist[v]
        if cum > rand:
            return v
```

If we want to generate multiple samples, repeatedly calling *sample_one* may not be efficient. If we want to generate *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)

```
def sample_multiple(dist, num_samples):
    """returns a list of num_samples values selected using distribution dist.
    dist is a {value:weight} dictionary that does not need to be normalized
    """
    total = sum(dist.values())
    rands = sorted(random.random()*total for i in range(num_samples))
    result = []
    dist_items = list(dist.items())
    cum = dist_items[0][1]  # cumulative sum
    index = 0
    for r in rands:
        while r>cum:
            index += 1
            cum += dist_items[index][1]
        result.append(dist_items[index][0])
    return result
```

Exercise 8.1

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

(a) *n* calls to *sample_one*

(b) *sample_multiple*

(c) Create the cumulative distribution (choose how this is represented) and, for each random number, do a binary search to determine the sample associated with the random number.

(d) Choose a random number in the range $[i/n, (i+1)/n)$ for each $i \in range(n)$, where *n* is the number of samples. Use these as the random numbers to select the particles. (Does this give random samples?)

For each method suggest when it might be the best method.

The *test_sampling* method can be used to generate the statistics from a number of samples. It is useful to see the variability as a function of the number of samples. Try it for few samples and also for many samples.

probStochSim.py — (continued)

```
def test_sampling(dist, num_samples):
```

```
    """Given a distribution, dist, draw num_samples samples
    and return the resulting counts
    """
    result = {v:0 for v in dist}
    for v in sample_multiple(dist, num_samples):
        result[v] += 1
    return result

# try the following queries a number of times each:
# test_sampling({1:1,2:2,3:3,4:4}, 100)
# test_sampling({1:1,2:2,3:3,4:4}, 100000)
```

8.8.2 Sampling Methods for Belief Network Inference

A *SamplingInferenceMethod* is an *InferenceMethod*, but the query method also takes arguments for the number of samples and the sample-order (which is an ordering of factors). The first methods assume a belief network (and not an undirected graphical model).

probStochSim.py — (continued)

```
class SamplingInferenceMethod(InferenceMethod):
    """The abstract class of sampling-based belief network inference methods"""
    def query(self,qvar,obs={},number_samples=1000,sample_order=None):
        raise NotImplementedError("SamplingInferenceMethod query") # abstract
```

8.8.3 Rejection Sampling

probStochSim.py — (continued)

```
class RejectionSampling(SamplingInferenceMethod):
    """The class that queries Graphical Models using Rejection Sampling.

    bn is a belief network to query
    """
    method_name = "rejection sampling"

    def __init__(self,bn=None):
        self.bn = bn

    def query(self,qvar,obs={},number_samples=1000,sample_order=None):
        """computes P(qvar|obs) where
        qvar is a variable.
        obs is a {variable:value} dictionary.
        sample_order is a list of variables where the parents
          come before the variable.
        """
        if sample_order is None:
            sample_order = self.bn.topological_sort()
        self.display(2,*sample_order,sep="\t")
```

```
        counts = {val:0 for val in qvar.domain}
        for i in range(number_samples):
            rejected = False
            sample = {}
            for nvar in sample_order:
                fac = self.bn.var2cpt[nvar]  #factor with nvar as child
                val = sample_one({v:fac.get_value(sample|{nvar:v}) for v in nvar.domain})
                self.display(2,val,end="\t")
                if nvar in obs and obs[nvar] != val:
                    rejected = True
                    self.display(2,"Rejected")
                    break
                sample[nvar] = val
            if not rejected:
                counts[sample[qvar]] += 1
                self.display(2,"Accepted")
        tot = sum(counts.values())
        # As well as the distribtion we also include raw counts
        return {c:v/tot if tot>0 else 1/len(qvar.domain) for (c,v) in counts.items()} | {"raw_count
```

8.8.4 Likelihood Weighting

Likelihood weighting includes a weight for each sample. Instead of rejecting samples based on observations, likelihood weighting changes the weights of the sample in proportion with the probability of the observation. The weight then becomes the probability that the variable would have been rejected.

probStochSim.py — (continued)

```
class LikelihoodWeighting(SamplingInferenceMethod):
    """The class that queries Graphical Models using Likelihood weighting.

    bn is a belief network to query
    """
    method_name = "likelihood weighting"

    def __init__(self,bn=None):
        self.bn = bn

    def query(self,qvar,obs={},number_samples=1000,sample_order=None):
        """computes P(qvar|obs) where
        qvar is a variable.
        obs is a {variable:value} dictionary.
        sample_order is a list of factors where factors defining the parents
          come before the factors for the child.
        """
        if sample_order is None:
            sample_order = self.bn.topological_sort()
        self.display(2,*[v for v in sample_order
                          if v not in obs],sep="\t")
        counts = {val:0 for val in qvar.domain}
        for i in range(number_samples):
```

```
            sample = {}
            weight = 1.0
            for nvar in sample_order:
                fac = self.bn.var2cpt[nvar]
                if nvar in obs:
                    sample[nvar] = obs[nvar]
                    weight *= fac.get_value(sample)
                else:
                    val = sample_one({v:fac.get_value(sample|{nvar:v}) for v in nvar.domain})
                    self.display(2,val,end="\t")
                    sample[nvar] = val
            counts[sample[qvar]] += weight
            self.display(2,weight)
        tot = sum(counts.values())
        # as well as the distribition we also include the raw counts
        return {c:v/tot for (c,v) in counts.items()} | {"raw_counts":counts}
```

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).

8.8.5 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)

```
class ParticleFiltering(SamplingInferenceMethod):
    """The class that queries Graphical Models using Particle Filtering.

    bn is a belief network to query
    """
    method_name = "particle filtering"

    def __init__(self,bn=None):
        self.bn = bn

    def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
        """computes P(qvar|obs) where
        qvar is a variable.
        obs is a {variable:value} dictionary.
        sample_order is a list of factors where factors defining the parents
          come before the factors for the child.
        """
        if sample_order is None:
            sample_order = self.bn.topological_sort()
        self.display(2,*[v for v in sample_order
```

```
                         if v not in obs],sep="\t")
        particles = [{} for i in range(number_samples)]
        for nvar in sample_order:
            fac = self.bn.var2cpt[nvar]
            if nvar in obs:
                weights = [fac.get_value(part|{nvar:obs[nvar]}) for part in particles]
                particles = [p|{nvar:obs[nvar]} for p in resample(particles, weights, number_sample
            else:
                for part in particles:
                    part[nvar] = sample_one({v:fac.get_value(part|{nvar:v}) for v in nvar.domain})
                self.display(2,part[nvar],end="\t")
        counts = {val:0 for val in qvar.domain}
        for part in particles:
            counts[part[qvar]] += 1
        tot = sum(counts.values())
        # as well as the distribution we also include the raw counts
        return {c:v/tot for (c,v) in counts.items()} | {"raw_counts":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)

```
def resample(particles, weights, num_samples):
    """returns num_samples copies of particles resampled according to weights.
    particles is a list of particles
    weights is a list of positive numbers, of same length as particles
    num_samples is n integer
    """
    total = sum(weights)
    rands = sorted(random.random()*total for i in range(num_samples))
    result = []
    cum = weights[0]   # cumulative sum
    index = 0
    for r in rands:
        while r>cum:
            index += 1
            cum += weights[index]
        result.append(particles[index])
    return result
```

8.8.6 Examples

probStochSim.py — (continued)

```
from probGraphicalModels import bn_4ch, A,B,C,D
bn_4chr = RejectionSampling(bn_4ch)
```

```
bn_4chL = LikelihoodWeighting(bn_4ch)
## InferenceMethod.max_display_level = 2 # detailed tracing for all inference methods
## bn_4chr.query(A,{})
## bn_4chr.query(C,{})
## bn_4chr.query(A,{C:True})
## bn_4chr.query(B,{A:True,C:False})

from probGraphicalModels import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
bn_reportr = RejectionSampling(bn_report) # answers queries using rejection sampling
bn_reportL = LikelihoodWeighting(bn_report) # answers queries using rejection sampling
bn_reportp = ParticleFiltering(bn_report) # answers queries using particle filtering
## bn_reportr.query(Tamper,{})
## bn_reportr.query(Tamper,{})
## bn_reportr.query(Tamper,{Report:True})
## InferenceMethod.max_display_level = 0 # no detailed tracing for all inference methods
## bn_reportr.query(Tamper,{Report:True},number_samples=100000)
## bn_reportr.query(Tamper,{Report:True,Smoke:False})
## bn_reportr.query(Tamper,{Report:True,Smoke:False},number_samples=100)

## bn_reportL.query(Tamper,{Report:True,Smoke:False},number_samples=100)
## bn_reportL.query(Tamper,{Report:True,Smoke:False},number_samples=100)


from probGraphicalModels import bn_sprinkler,Season, Sprinkler
from probGraphicalModels import Rained, Grass_wet, Grass_shiny, Shoes_wet
bn_sprinklerr = RejectionSampling(bn_sprinkler) # answers queries using rejection sampling
bn_sprinklerL = LikelihoodWeighting(bn_sprinkler) # answers queries using rejection sampling
bn_sprinklerp = ParticleFiltering(bn_sprinkler) # answers queries using particle filtering
#bn_sprinklerr.query(Shoes_wet,{Grass_shiny:True,Rained:True})
#bn_sprinklerL.query(Shoes_wet,{Grass_shiny:True,Rained:True})
#bn_sprinklerp.query(Shoes_wet,{Grass_shiny:True,Rained:True})

if __name__ == "__main__":
    InferenceMethod.testIM(RejectionSampling, threshold=0.1)
    InferenceMethod.testIM(LikelihoodWeighting, threshold=0.1)
    InferenceMethod.testIM(LikelihoodWeighting, threshold=0.1)
```

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.

8.8.7 Gibbs Sampling

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

probStochSim.py — (continued)

```
#import random
#from probGraphicalModels import InferenceMethod

#from probStochSim import sample_one, SamplingInferenceMethod

class GibbsSampling(SamplingInferenceMethod):
    """The class that queries Graphical Models using Gibbs Sampling.

    bn is a graphical model (e.g., a belief network) to query
    """
    method_name = "Gibbs sampling"

    def __init__(self,bn=None):
        self.bn = bn

    def query(self, qvar, obs={}, number_samples=1000, burn_in=100, sample_order=None):
        """computes P(qvar|obs) where
        qvar is a variable.
        obs is a {variable:value} dictionary.
        sample_order is a list of non-observed variables in order, or
        if sample_order None, the variables are shuffled at each iteration.
        """
        counts = {val:0 for val in qvar.domain}
        if sample_order is not None:
            variables = sample_order
        else:
            variables = [v for v in self.bn.variables if v not in obs]
        var_to_factors = {v:set() for v in self.bn.variables}
        for fac in self.bn.factors:
            for var in fac.variables:
                var_to_factors[var].add(fac)
        sample = {var:random.choice(var.domain) for var in variables}
        self.display(2,"Sample:",sample)
        sample.update(obs)
        for i in range(burn_in + number_samples):
            if sample_order == None:
                random.shuffle(variables)
            for var in variables:
                # get unnormalized probability distribution of var given its neighbours
                vardist = {val:1 for val in var.domain}
                for val in var.domain:
                    sample[var] = val
                    for fac in var_to_factors[var]: # Markov blanket
                        vardist[val] *= fac.get_value(sample)
                sample[var] = sample_one(vardist)
            if i >= burn_in:
                counts[sample[qvar]] +=1
        tot = sum(counts.values())
        # as well as the computed distribution, we also include raw counts
        return {c:v/tot for (c,v) in counts.items()} | {"raw_counts":counts}
```

```python

#from probGraphicalModels import bn_4ch, A,B,C,D
bn_4chg = GibbsSampling(bn_4ch)
## InferenceMethod.max_display_level = 2 # detailed tracing for all inference methods
bn_4chg.query(A,{})
## bn_4chg.query(D,{})
## bn_4chg.query(B,{D:True})
## bn_4chg.query(B,{A:True,C:False})

from probGraphicalModels import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
bn_reportg = GibbsSampling(bn_report)
## bn_reportg.query(Tamper,{Report:True},number_samples=1000)

if __name__ == "__main__":
    InferenceMethod.testIM(GibbsSampling, threshold=0.1)
```

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?

8.8.8 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)

```
import matplotlib.pyplot as plt

def plot_stats(method, qvar, qval, obs, number_runs=1000, **queryargs):
    """Plots a cumulative distribution of the prediction of the model.
    method is a InferenceMethod (that implements appropriate query(.))
    plots P(qvar=qval | obs)
    qvar is the query variable, qval is corresponding value
    obs is the {variable:value} dictionary representing the observations
    number_iterations is the number of runs that are plotted
    **queryargs is the arguments to query (often number_samples for sampling methods)
    """
    plt.ion()
    plt.xlabel("value")
    plt.ylabel("Cumulative Number")
    method.max_display_level, prev_mdl = 0, method.max_display_level #no display
    answers = [method.query(qvar,obs,**queryargs)
               for i in range(number_runs)]
    values = [ans[qval] for ans in answers]
    label = f"{method.method_name} P({qvar}={qval}|{','.join(f'{var}={val}' for (var,val) in obs.i
    values.sort()
    plt.plot(values,range(number_runs),label=label)
    plt.legend() #loc="upper left")
    plt.draw()
    method.max_display_level = prev_mdl # restore display level

# Try:
# plot_stats(bn_reportr,Tamper,True,{Report:True,Smoke:True},number_samples=1000, number_runs=1000
# plot_stats(bn_reportL,Tamper,True,{Report:True,Smoke:True},number_samples=1000, number_runs=1000
# plot_stats(bn_reportp,Tamper,True,{Report:True,Smoke:True},number_samples=1000, number_runs=1000
# plot_stats(bn_reportr,Tamper,True,{Report:True,Smoke:True},number_samples=100, number_runs=1000)
# plot_stats(bn_reportL,Tamper,True,{Report:True,Smoke:True},number_samples=100, number_runs=1000)
# plot_stats(bn_reportg,Tamper,True,{Report:True,Smoke:True},number_samples=1000, number_runs=1000

def plot_mult(methods, example, qvar, qval, obs, number_samples=1000, number_runs=1000):
    for method in methods:
        solver = method(example)
        if isinstance(method,SamplingInferenceMethod):
            plot_stats(solver, qvar, qval, obs, number_samples, number_runs)
        else:
            plot_stats(solver, qvar, qval, obs, number_runs)

from probRC import RC
# Try following (but it takes a while..)
methods = [RC,RejectionSampling,LikelihoodWeighting,ParticleFiltering,GibbsSampling]
#plot_mult(methods,bn_report,Tamper,True,{Report:True,Smoke:False},number_samples=100, number_runs
# plot_mult(methods,bn_report,Tamper,True,{Report:True,Smoke:True},number_samples=100, number_runs

# Sprinkler Example:
# plot_stats(bn_sprinklerr,Shoes_wet,True,{Grass_shiny:True,Rained:True},number_samples=1000)
```

```
# plot_stats(bn_sprinklerL,Shoes_wet,True,{Grass_shiny:True,Rained:True},number_samples=1000)
```

8.9 Hidden Markov Models

This code for hidden Markov models is independent of the graphical models code, to keep it simple. Section 8.10 gives code that models hidden Markov models, and more generally, dynamic belief networks, using the graphical models code.

This HMM code assumes there are multiple Boolean observation variables that depend on the current state and are independent of each other given the state.

probHMM.py — Hidden Markov Model

```
import random
from probStochSim import sample_one, sample_multiple

class HMM(object):
    def __init__(self, states, obsvars,pobs,trans,indist):
        """A hidden Markov model.
        states - set of states
        obsvars - set of observation variables
        pobs - probability of observations, pobs[i][s] is P(Obs_i=True | State=s)
        trans - transition probability - trans[i][j] gives P(State=j | State=i)
        indist - initial distribution - indist[s] is P(State_0 = s)
        """
        self.states = states
        self.obsvars = obsvars
        self.pobs = pobs
        self.trans = trans
        self.indist = indist
```

Consider the following example. Suppose you want to unobtrusively keep track of an animal in a triangular enclosure using sound. Suppose you have 3 microphones that provide unreliable (noisy) binary information at each time step. The animal is either close to one of the 3 points of the triangle or in the middle of the triangle.

probHMM.py — (continued)

```
# state
#       0=middle, 1,2,3 are corners
states1 = {'middle', 'c1', 'c2', 'c3'} # states
obs1 = {'m1','m2','m3'} # microphones
```

The observation model is as follows. If the animal is in a corner, it will be detected by the microphone at that corner with probability 0.6, and will be independently detected by each of the other microphones with a probability of 0.1. If the animal is in the middle, it will be detected by each microphone with a probability of 0.4.

probHMM.py — (continued)

```
# pobs gives the observation model:
#pobs[mi][state] is P(mi=on | state)
closeMic=0.6; farMic=0.1; midMic=0.4
pobs1 = {'m1':{'middle':midMic, 'c1':closeMic, 'c2':farMic, 'c3':farMic}, # mic 1
        'm2':{'middle':midMic, 'c1':farMic, 'c2':closeMic, 'c3':farMic}, # mic 2
        'm3':{'middle':midMic, 'c1':farMic, 'c2':farMic, 'c3':closeMic}} # mic 3
```

The transition model is as follows: If the animal is in a corner it stays in the same corner with probability 0.80, goes to the middle with probability 0.1 or goes to one of the other corners with probability 0.05 each. If it is in the middle, it stays in the middle with probability 0.7, otherwise it moves to one the corners, each with probability 0.1.

probHMM.py — (continued)

```
# trans specifies the dynamics
# trans[i] is the distribution over states resulting from state i
# trans[i][j] gives P(S=j | S=i)
sm=0.7; mmc=0.1            # transition probabilities when in middle
sc=0.8; mcm=0.1; mcc=0.05   # transition probabilities when in a corner
trans1 = {'middle':{'middle':sm, 'c1':mmc, 'c2':mmc, 'c3':mmc}, # was in middle
         'c1':{'middle':mcm, 'c1':sc, 'c2':mcc, 'c3':mcc}, # was in corner 1
         'c2':{'middle':mcm, 'c1':mcc, 'c2':sc, 'c3':mcc}, # was in corner 2
         'c3':{'middle':mcm, 'c1':mcc, 'c2':mcc, 'c3':sc}} # was in corner 3
```

Initially the animal is in one of the four states, with equal probability.

probHMM.py — (continued)

```
# initially we have a uniform distribution over the animal's state
indist1 = {st:1.0/len(states1) for st in states1}

hmm1 = HMM(states1, obs1, pobs1, trans1, indist1)
```

8.9.1 Exact Filtering for HMMs

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

probHMM.py — (continued)

```
from display import Displayable

class HMMVEfilter(Displayable):
    def __init__(self,hmm):
        self.hmm = hmm
        self.state_dist = hmm.indist

    def filter(self, obsseq):
        """updates and returns the state distribution following the sequence of
        observations in obsseq using variable elimination.

```

```
        Note that it first advances time.
        This is what is required if it is called sequentially.
        If that is not what is wanted initially, do an observe first.
        """
        for obs in obsseq:
            self.advance()    # advance time
            self.observe(obs) # observe
        return self.state_dist

    def observe(self, obs):
        """updates state conditioned on observations.
        obs is a list of values for each observation variable"""
        for i in self.hmm.obsvars:
            self.state_dist = {st:self.state_dist[st]*(self.hmm.pobs[i][st]
                                          if obs[i] else (1-self.hmm.pobs[i][st]))
                               for st in self.hmm.states}
        norm = sum(self.state_dist.values()) # normalizing constant
        self.state_dist = {st:self.state_dist[st]/norm for st in self.hmm.states}
        self.display(2,"After observing",obs,"state distribution:",self.state_dist)

    def advance(self):
        """advance to the next time"""
        nextstate = {st:0.0 for st in self.hmm.states}  # distribution over next states
        for j in self.hmm.states:     # j ranges over next states
            for i in self.hmm.states: # i ranges over previous states
                nextstate[j] += self.hmm.trans[i][j]*self.state_dist[i]
        self.state_dist = nextstate
```

The following are some queries for *hmm*1.

probHMM.py — (continued)

```
hmm1f1 = HMMVEfilter(hmm1)
# hmm1f1.filter([{'m1':0, 'm2':1, 'm3':1}, {'m1':1, 'm2':0, 'm3':1}])
## HMMVEfilter.max_display_level = 2 # show more detail in displaying
# hmm1f2 = HMMVEfilter(hmm1)
# hmm1f2.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':1, 'm3':0}, {'m1':1, 'm2':0, 'm3':0},
#               {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
#               {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':1}, {'m1':0, 'm2':0, 'm3':1},
#               {'m1':0, 'm2':0, 'm3':1}])
# hmm1f3 = HMMVEfilter(hmm1)
# hmm1f3.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':1, 'm2':0, 'm3':0}, {'m1':

# How do the following differ in the resulting state distribution?
# Note they start the same, but have different initial observations.
## HMMVEfilter.max_display_level = 1 # show less detail in displaying
# for i in range(100): hmm1f1.advance()
# hmm1f1.state_dist
# for i in range(100): hmm1f3.advance()
# 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.

8.9.2 Particle Filtering for HMMs

In this implementation a particle is just a state. If you want to do some form of smoothing, a particle should probably be a history of states. This maintains, *particles*, an array of states, *weights* an array of (non-negative) real numbers, such that *weights*[*i*] is the weight of *particles*[*i*].

probHMM.py — (continued)

```
from display import Displayable
from probStochSim import resample

class HMMparticleFilter(Displayable):
    def __init__(self,hmm,number_particles=1000):
        self.hmm = hmm
        self.particles = [sample_one(hmm.indist)
                          for i in range(number_particles)]
        self.weights = [1 for i in range(number_particles)]

    def filter(self, obsseq):
        """returns the state distribution following the sequence of
        observations in obsseq using particle filtering.

        Note that it first advances time.
        This is what is required if it is called after previous filtering.
        If that is not what is wanted initially, do an observe first.
        """
        for obs in obsseq:
            self.advance()   # advance time
            self.observe(obs) # observe
            self.resample_particles()
            self.display(2,"After observing", str(obs),
                           "state distribution:", self.histogram(self.particles))
        self.display(1,"Final state distribution:", self.histogram(self.particles))
        return self.histogram(self.particles)

    def advance(self):
        """advance to the next time.
        This assumes that all of the weights are 1."""
        self.particles = [sample_one(self.hmm.trans[st])
                          for st in self.particles]

```

```
    def observe(self, obs):
        """reweighs the particles to incorporate observations obs"""
        for i in range(len(self.particles)):
            for obv in obs:
                if obs[obv]:
                    self.weights[i] *= self.hmm.pobs[obv][self.particles[i]]
                else:
                    self.weights[i] *= 1-self.hmm.pobs[obv][self.particles[i]]

    def histogram(self, particles):
        """returns list of the probability of each state as represented by
        the particles"""
        tot=0
        hist = {st: 0.0 for st in self.hmm.states}
        for (st,wt) in zip(self.particles,self.weights):
            hist[st]+=wt
            tot += wt
        return {st:hist[st]/tot for st in hist}

    def resample_particles(self):
        """resamples to give a new set of particles."""
        self.particles = resample(self.particles, self.weights, len(self.particles))
        self.weights = [1] * len(self.particles)
```

The following are some queries for *hmm*1.

probHMM.py — (continued)

```
hmm1pf1 = HMMparticleFilter(hmm1)
# HMMparticleFilter.max_display_level = 2 # show each step
# hmm1pf1.filter([{'m1':0, 'm2':1, 'm3':1}, {'m1':1, 'm2':0, 'm3':1}])
# hmm1pf2 = HMMparticleFilter(hmm1)
# hmm1pf2.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':1, 'm3':0}, {'m1':1, 'm2':0, 'm3':0},
#                 {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
#                 {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':1}, {'m1':0, 'm2':0, 'm3':1},
#                 {'m1':0, 'm2':0, 'm3':1}])
# hmm1pf3 = HMMparticleFilter(hmm1)
# hmm1pf3.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':1, 'm2':0, 'm3':0}, {'m1'
```

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.

8.9.3 Generating Examples

The following code is useful for generating examples.

probHMM.py — (continued)

```
def simulate(hmm,horizon):
    """returns a pair of (state sequence, observation sequence) of length horizon.
    for each time t, the agent is in state_sequence[t] and
    observes observation_sequence[t]
    """
    state = sample_one(hmm.indist)
    obsseq=[]
    stateseq=[]
    for time in range(horizon):
        stateseq.append(state)
        newobs = {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
                  for obs in hmm.obsvars}
        obsseq.append(newobs)
        state = sample_one(hmm.trans[state])
    return stateseq,obsseq

def simobs(hmm,stateseq):
    """returns observation sequence for the state sequence"""
    obsseq=[]
    for state in stateseq:
        newobs = {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
                  for obs in hmm.obsvars}
        obsseq.append(newobs)
    return obsseq

def create_eg(hmm,n):
    """Create an annotated example for horizon n"""
    seq,obs = simulate(hmm,n)
    print("True state sequence:",seq)
    print("Sequence of observations:\n",obs)
    hmmfilter = HMMVEfilter(hmm)
    dist = hmmfilter.filter(obs)
    print("Resulting distribution over states:\n",dist)
```

8.10 Dynamic Belief Networks

A **dynamic belief network (DBN)** is a belief network that extends in time.

There are a number of ways that reasoning can be carried out in a DBN, including:

- Rolling out the DBN for some time period, and using standard belief network inference. The latest time that needs to be in the rolled out network is the time of the latest observation or the time of a query (whichever is later). This allows us to observe any variables at any time and query any variables at any time. This is covered in Section 8.10.2.

- An unrolled belief network may be very large, and we might only be interested in asking about "now". In this case we can just representing the variables "now". In this approach we can observe and query the current variables. We can them move to the next time. This does not allow for arbitrary historical queries (about the past or the future), but can be much simpler. This is covered in Section 8.10.3.

8.10.1 Representing Dynamic Belief Networks

To specify a DBN, think about the distribution *now*. *Now* will be represented as time 1. Each variable will have a corresponding previous variable; these will be created together.

A dynamic belief network consists of:

- A set of features. A variable is a feature-time pair.
- An initial distribution over the features "now" (time 1). This is a belief network with all variables being time 1 variables.
- A specification of the dynamics. We define the how the variables *now* (time 1) depend on variables *now* and the previous time (time 0), in such a way that the graph is acyclic.

probDBN.py — Dynamic belief networks

```
from probVariables import Variable
from probGraphicalModels import GraphicalModel, BeliefNetwork
from probFactors import Prob
from probVE import VE
from display import Displayable

class DBNvariable(Variable):
    """A random variable that incorporates the stage (time)

    A variable can have both a name and an index. The index defaults to 1.
    """
    def __init__(self,name,domain=[False,True],index=1):
        Variable.__init__(self,f"{name}_{index}",domain)
        self.basename = name
        self.domain = domain
        self.index = index
        self.previous = None

    def __lt__(self,other):
        if self.name != other.name:
            return self.name<other.name
```

```
        else:
            return self.index<other.index

    def __gt__(self,other):
        return other<self

def variable_pair(name,domain=[False,True]):
    """returns a variable and its predecessor. This is used to define 2-stage DBNs

    If the name is X, it returns the pair of variables X_prev,X_now"""
    var_now = DBNvariable(name,domain,index='now')
    var_prev = DBNvariable(name,domain,index='prev')
    var_now.previous = var_prev
    return var_prev, var_now
```

A *FactorRename* 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.

probDBN.py — (continued)

```
class FactorRename(Factor):
    def __init__(self,fac,renaming):
        """A renamed factor.
        fac is a factor
        renaming is a dictionary of the form {new:old} where old and new var variables,
           where the variables in fac appear exactly once in the renaming
        """
        Factor.__init__(self,[n for (n,o) in renaming.items() if o in fac.variables])
        self.orig_fac = fac
        self.renaming = renaming

    def get_value(self,assignment):
        return self.orig_fac.get_value({self.renaming[var]:val
                                        for (var,val) in assignment.items()
                                        if var in self.variables})
```

The following class renames the variables of a conditional probability distribution. It is used for template models (e.g., dynamic decision networks or relational models)

probDBN.py — (continued)

```
class CPDrename(FactorRename, CPD):
    def __init__(self, cpd, renaming):
        renaming_inverse = {old:new for (new,old) in renaming.items()}
        CPD.__init__(self,renaming_inverse[cpd.child],[renaming_inverse[p] for p in cpd.parents])
        self.orig_fac = cpd
        self.renaming = renaming
```

probDBN.py — (continued)

```
class DBN(Displayable):
```

```
    """The class of stationary Dynamic Belief networks.
    * name is the DBN name
    * vars_now is a list of current variables (each must have
    previous variable).
    * transition_factors is a list of factors for P(X|parents) where X
    is a current variable and parents is a list of current or previous variables.
    * init_factors is a list of factors for P(X|parents) where X is a
    current variable and parents can only include current variables
    The graph of transition factors + init factors must be acyclic.

    """
    def __init__(self, title, vars_now, transition_factors=None, init_factors=None):
        self.title = title
        self.vars_now = vars_now
        self.vars_prev = [v.previous for v in vars_now]
        self.transition_factors = transition_factors
        self.init_factors = init_factors
        self.var_index = {}    # var_index[v] is the index of variable v
        for i,v in enumerate(vars_now):
            self.var_index[v]=i
```

Here is a 3 variable DBN:

probDBN.py — (continued)

```
A0,A1 = variable_pair("A", domain=[False,True])
B0,B1 = variable_pair("B", domain=[False,True])
C0,C1 = variable_pair("C", domain=[False,True])

# dynamics
pc = Prob(C1,[B1,C0],[[[0.03,0.97],[0.38,0.62]],[[0.23,0.77],[0.78,0.22]]])
pb = Prob(B1,[A0,A1],[[[0.5,0.5],[0.77,0.23]],[[0.4,0.6],[0.83,0.17]]])
pa = Prob(A1,[A0,B0],[[[0.1,0.9],[0.65,0.35]],[[0.3,0.7],[0.8,0.2]]])

# initial distribution
pa0 = Prob(A1,[],[0.9,0.1])
pb0 = Prob(B1,[A1],[[0.3,0.7],[0.8,0.2]])
pc0 = Prob(C1,[],[0.2,0.8])

db1 = DBN("Simple DBN",[A1,B1,C1],[pa,pb,pc],[pa0,pb0,pc0])
```

Here is the animal example

probDBN.py — (continued)

```
from probHMM import closeMic, farMic, midMic, sm, mmc, sc, mcm, mcc

Pos_0,Pos_1 = variable_pair("Position",domain=[0,1,2,3])
Mic1_0,Mic1_1 = variable_pair("Mic1")
Mic2_0,Mic2_1 = variable_pair("Mic2")
Mic3_0,Mic3_1 = variable_pair("Mic3")

# conditional probabilities - see hmm for the values of sm,mmc, etc
ppos = Prob(Pos_1, [Pos_0],
```

```
          [[sm, mmc, mmc, mmc], #was in middle
           [mcm, sc, mcc, mcc], #was in corner 1
           [mcm, mcc, sc, mcc], #was in corner 2
           [mcm, mcc, mcc, sc]]) #was in corner 3
pm1 = Prob(Mic1_1, [Pos_1], [[1-midMic, midMic], [1-closeMic, closeMic],
                         [1-farMic, farMic], [1-farMic, farMic]])
pm2 = Prob(Mic2_1, [Pos_1], [[1-midMic, midMic], [1-farMic, farMic],
                         [1-closeMic, closeMic], [1-farMic, farMic]])
pm3 = Prob(Mic3_1, [Pos_1], [[1-midMic, midMic], [1-farMic, farMic],
                         [1-farMic, farMic], [1-closeMic, closeMic]])
ipos = Prob(Pos_1,[], [0.25, 0.25, 0.25, 0.25])
dbn_an =DBN("Animal DBN",[Pos_1,Mic1_1,Mic2_1,Mic3_1],
           [ppos, pm1, pm2, pm3],
           [ipos, pm1, pm2, pm3])
```

8.10.2 Unrolling DBNs

probDBN.py — (continued)

```
class BNfromDBN(BeliefNetwork):
    """Belief Network unrolled from a dynamic belief network
    """

    def __init__(self,dbn,horizon):
        """dbn is the dynamic belief network being unrolled
        horizon>0 is the number of steps (so there will be horizon+1 variables for each DBN variabl
        """
        self.name2var = {var.basename: [DBNvariable(var.basename,var.domain,index) for index in ran
                         for var in dbn.vars_now}
        self.display(1,f"name2var={self.name2var}")
        variables = {v for vs in self.name2var.values() for v in vs}
        self.display(1,f"variables={variables}")
        bnfactors = {CPDrename(fac,{self.name2var[var.basename][0]:var
                                       for var in fac.variables})
                      for fac in dbn.init_factors}
        bnfactors |= {CPDrename(fac,{self.name2var[var.basename][i]:var
                                       for var in fac.variables if var.index=='prev'}
                                 | {self.name2var[var.basename][i+1]:var
                                       for var in fac.variables if var.index=='now'})
                      for fac in dbn.transition_factors
                          for i in range(horizon)}
        self.display(1,f"bnfactors={bnfactors}")
        BeliefNetwork.__init__(self, dbn.title, variables, bnfactors)
```

Here are two examples. Note that we need to use `bn.name2var['B'][2]` to get the variable B2 (B at time 2).

probDBN.py — (continued)

```
# Try
#from probRC import RC
```

```
#bn = BNfromDBN(dbn1,2) # construct belief network
#drc = RC(bn)           # initialize recursive conditioning
#B2 = bn.name2var['B'][2]
#drc.query(B2) #P(B2)
#drc.query(bn.name2var['B'][1],{bn.name2var['B'][0]:True,bn.name2var['C'][1]:False}) #P(B1|B0,C1)
```

8.10.3 DBN Filtering

If we only wanted to ask questions about the current state, we can save space by forgetting the history variables.

probDBN.py — (continued)

```
class DBNVEfilter(VE):
    def __init__(self,dbn):
        self.dbn = dbn
        self.current_factors = dbn.init_factors
        self.current_obs = {}

    def observe(self, obs):
        """updates the current observations with obs.
        obs is a variable:value dictionary where variable is a current
        variable.
        """
        assert all(self.current_obs[var]==obs[var] for var in obs
                   if var in self.current_obs),"inconsistent current observations"
        self.current_obs.update(obs) # note 'update' is a dict method

    def query(self,var):
        """returns the posterior probability of current variable var"""
        return VE(GraphicalModel(self.dbn.title,self.dbn.vars_now,self.current_factors)).query(var,sel

    def advance(self):
        """advance to the next time"""
        prev_factors = [self.make_previous(fac) for fac in self.current_factors]
        prev_obs = {var.previous:val for var,val in self.current_obs.items()}
        two_stage_factors = prev_factors + self.dbn.transition_factors
        self.current_factors = self.elim_vars(two_stage_factors,self.dbn.vars_prev,prev_obs)
        self.current_obs = {}

    def make_previous(self,fac):
         """Creates new factor from fac where the current variables in fac
         are renamed to previous variables.
         """
         return FactorRename(fac, {var.previous:var for var in fac.variables})

    def elim_vars(self,factors, vars, obs):
        for var in vars:
            if var in obs:
                factors = [self.project_observations(fac,obs) for fac in factors]
            else:
                factors = self.eliminate_var(factors, var)
```

```
        return factors
```

Example queries:

probDBN.py — (continued)

```
#df = DBNVEfilter(dbn1)
#df.observe({B1:True}); df.advance(); df.observe({C1:False})
#df.query(B1) #P(B1|B0,C1)
#df.advance(); df.query(B1)
#dfa = DBNVEfilter(dbn_an)
# dfa.observe({Mic1_1:0, Mic2_1:1, Mic3_1:1})
# dfa.advance()
# dfa.observe({Mic1_1:1, Mic2_1:0, Mic3_1:1})
# dfa.query(Pos_1)
```

Chapter 9

Planning with 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*. In a **utility table** the utility is defined in terms of a, a list that enumerates the values as in Section 8.3.3.

decnNetworks.py — Representations for Decision Networks

```
from probGraphicalModels import GraphicalModel, BeliefNetwork
from probFactors import Factor, CPD, TabFactor, factor_times, Prob
from probVariables import Variable
import matplotlib.pyplot as plt

class Utility(Factor):
    """A factor defining a utility"""
    pass

class UtilityTable(TabFactor, Utility):
    """A factor defining a utility using a table"""
    def __init__(self, vars, table, position=None):
        """Creates a factor on vars from the table.
        The table is ordered according to vars.
        """
        TabFactor.__init__(self,vars,table)
        self.position = position
```

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. It also includes a potion, which is only used for plotting.

decnNetworks.py — (continued)

```
class DecisionVariable(Variable):
    def __init__(self, name, domain, parents, position=None):
        Variable.__init__(self, name, domain, position)
        self.parents = parents
        self.all_vars = set(parents) | {self}
```

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

decnNetworks.py — (continued)

```
class DecisionNetwork(BeliefNetwork):
    def __init__(self, title, vars, factors):
        """vars is a list of variables
        factors is a list of factors (instances of CPD and Utility)
        """
        GraphicalModel.__init__(self, title, vars, factors) # don't call init for BeliefNetwork
        self.var2parents = ({v : v.parents for v in vars if isinstance(v,DecisionVariable)}
                    | {f.child:f.parents for f in factors if isinstance(f,CPD)})
        self.children = {n:[] for n in self.variables}
        for v in self.var2parents:
            for par in self.var2parents[v]:
                self.children[par].append(v)
        self.utility_factor = [f for f in factors if isinstance(f,Utility)][0]
        self.topological_sort_saved = None
```

The split order ensures that the parents of a decision node are split before the decision node, and no other variables (if that is possible).

decnNetworks.py — (continued)

```
    def split_order(self):
        so = []
        tops = self.topological_sort()
        for v in tops:
            if isinstance(v,DecisionVariable):
               so += [p for p in v.parents if p not in so]
               so.append(v)
        so += [v for v in tops if v not in so]
        return so
```

decnNetworks.py — (continued)

```
    def show(self):
        plt.ion()  # interactive
        ax = plt.figure().gca()
        ax.set_axis_off()
        plt.title(self.title)
        for par in self.utility_factor.variables:
            ax.annotate("Utility", par.position, xytext=self.utility_factor.position,
                                    arrowprops={'arrowstyle':'<-'},bbox=dict(boxstyle="sawtooth,pad=1.0",
                                    ha='center')
```

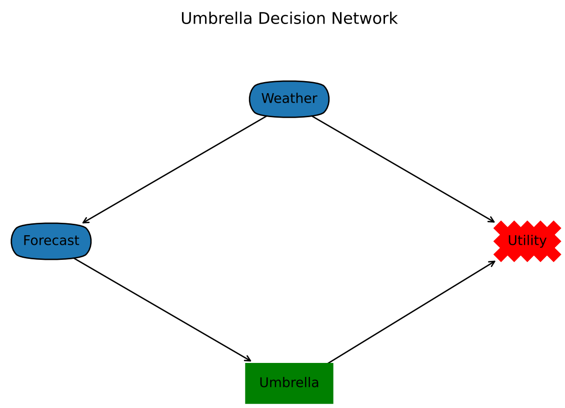


Figure 9.1: The umbrella decision network

```
        for var in reversed(self.topological_sort()):
            if isinstance(var,DecisionVariable):
                bbox = dict(boxstyle="square,pad=1.0",color="green")
            else:
               bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5")
            if self.var2parents[var]:
                for par in self.var2parents[var]:
                    ax.annotate(var.name, par.position, xytext=var.position,
                                    arrowprops={'arrowstyle':'<-'},bbox=bbox,
                                    ha='center')
            else:
                x,y = var.position
                plt.text(x,y,var.name,bbox=bbox,ha='center')
```

9.1.1 Example Decision Networks

Umbrella Decision Network

Here is a simple "umbrella" decision network. The output of `umbrella_dn.show()` is shown in Figure 9.1.

decnNetworks.py — (continued)

```
Weather = Variable("Weather", ["NoRain", "Rain"], position=(0.5,0.8))
Forecast = Variable("Forecast", ["Sunny", "Cloudy", "Rainy"], position=(0,0.4))
# Each variant uses one of the following:
Umbrella = DecisionVariable("Umbrella", ["Take", "Leave"], {Forecast}, position=(0.5,0))

p_weather = Prob(Weather, [], [0.7, 0.3])
p_forecast = Prob(Forecast, [Weather], [[0.7, 0.2, 0.1], [0.15, 0.25, 0.6]])
umb_utility = UtilityTable([Weather, Umbrella], [[20, 100], [70, 0]], position=(1,0.4))

```

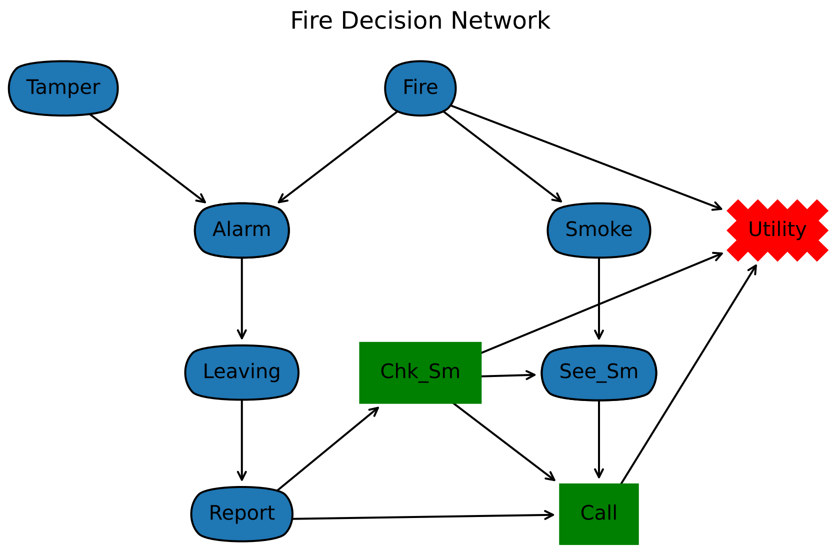


Figure 9.2: Fire Decision Network

```
umbrella_dn = DecisionNetwork("Umbrella Decision Network",
                              {Weather, Forecast, Umbrella},
                              {p_weather, p_forecast, umb_utility})
```

The following is a variant with the umbrella decision having 2 parents; nothing else has changed. This is interesting because one of the parents is not needed; if the agent knows the weather, it can ignore the forecast.

decnNetworks.py — (continued)

```
Umbrella2p = DecisionVariable("Umbrella", ["Take", "Leave"], {Forecast, Weather}, position=(0.5,0))
umb_utility2p = UtilityTable([Weather, Umbrella2p], [[20, 100], [70, 0]], position=(1,0.4))
umbrella_dn2p = DecisionNetwork("Umbrella Decision Network (extra arc)",
                              {Weather, Forecast, Umbrella2p},
                              {p_weather, p_forecast, umb_utility2p})
```

Fire Decision Network

The fire decision network of Figure 9.2 (showing the result of `fire_dn.show()`) is represented as:

decnNetworks.py — (continued)

```
boolean = [False, True]
Alarm = Variable("Alarm", boolean, position=(0.25,0.633))
Fire = Variable("Fire", boolean, position=(0.5,0.9))
```

```
Leaving = Variable("Leaving", boolean, position=(0.25,0.366))
Report = Variable("Report", boolean, position=(0.25,0.1))
Smoke = Variable("Smoke", boolean, position=(0.75,0.633))
Tamper = Variable("Tamper", boolean, position=(0,0.9))

See_Sm = Variable("See_Sm", boolean, position=(0.75,0.366) )
Chk_Sm = DecisionVariable("Chk_Sm", boolean, {Report}, position=(0.5, 0.366))
Call = DecisionVariable("Call", boolean,{See_Sm,Chk_Sm,Report}, position=(0.75,0.1))

f_ta = Prob(Tamper,[],[0.98,0.02])
f_fi = Prob(Fire,[],[0.99,0.01])
f_sm = Prob(Smoke,[Fire],[[0.99,0.01],[0.1,0.9]])
f_al = Prob(Alarm,[Fire,Tamper],[[[0.9999, 0.0001], [0.15, 0.85]], [[0.01, 0.99], [0.5, 0.5]]])
f_lv = Prob(Leaving,[Alarm],[[0.999, 0.001], [0.12, 0.88]])
f_re = Prob(Report,[Leaving],[[0.99, 0.01], [0.25, 0.75]])
f_ss = Prob(See_Sm,[Chk_Sm,Smoke],[[[1,0],[1,0]],[[1,0],[0,1]]])

ut = UtilityTable([Chk_Sm,Fire,Call],[[[0,-200],[-5000,-200]],[[-20,-220],[-5020,-220]]], positior

fire_dn = DecisionNetwork("Fire Decision Network",
                        {Tamper,Fire,Alarm,Leaving,Smoke,Call,See_Sm,Chk_Sm,Report},
                        {f_ta,f_fi,f_sm,f_al,f_lv,f_re,f_ss,ut})
```

Cheating Decision Network

The following is the representation of the cheating decision of Figure 9.3. Note that we keep the names of the variables short (less than 8 characters) so that the tables look good when printed.

decnNetworks.py — (continued)

```
grades = ['A','B','C','F']
Watched = Variable("Watched", boolean, position=(0,0.9))
Caught1 = Variable("Caught1", boolean, position=(0.2,0.7))
Caught2 = Variable("Caught2", boolean, position=(0.6,0.7))
Punish = Variable("Punish", ["None","Suspension","Recorded"], position=(0.8,0.9))
Grade_1 = Variable("Grade_1", grades, position=(0.2,0.3))
Grade_2 = Variable("Grade_2", grades, position=(0.6,0.3))
Fin_Grd = Variable("Fin_Grd", grades, position=(0.8,0.1))
Cheat_1 = DecisionVariable("Cheat_1", boolean, set(), position=(0,0.5)) #no parents
Cheat_2 = DecisionVariable("Cheat_2", boolean, {Cheat_1,Caught1}, position=(0.4,0.5))

p_wa = Prob(Watched,[],[0.7, 0.3])
p_cc1 = Prob(Caught1,[Watched,Cheat_1],[[[1.0, 0.0], [0.9, 0.1]], [[1.0, 0.0], [0.5, 0.5]]])
p_cc2 = Prob(Caught2,[Watched,Cheat_2],[[[1.0, 0.0], [0.9, 0.1]], [[1.0, 0.0], [0.5, 0.5]]])
p_pun = Prob(Punish,[Caught1,Caught2],[[[1.0, 0.0, 0.0], [0.5, 0.4, 0.1]], [[0.6, 0.2, 0.2], [0.2,
p_gr1 = Prob(Grade_1,[Cheat_1], [{'A':0.2, 'B':0.3, 'C':0.3, 'D': 0.2}, {'A':0.5, 'B':0.3, 'C':0.2
p_gr2 = Prob(Grade_2,[Cheat_2], [{'A':0.2, 'B':0.3, 'C':0.3, 'D': 0.2}, {'A':0.5, 'B':0.3, 'C':0.2
p_fg = Prob(Fin_Grd,[Grade_1,Grade_2],
       {'A':{'A':{'A':1.0, 'B':0.0, 'C': 0.0, 'D':0.0},
             'B': {'A':0.5, 'B':0.5, 'C': 0.0, 'D':0.0},
```

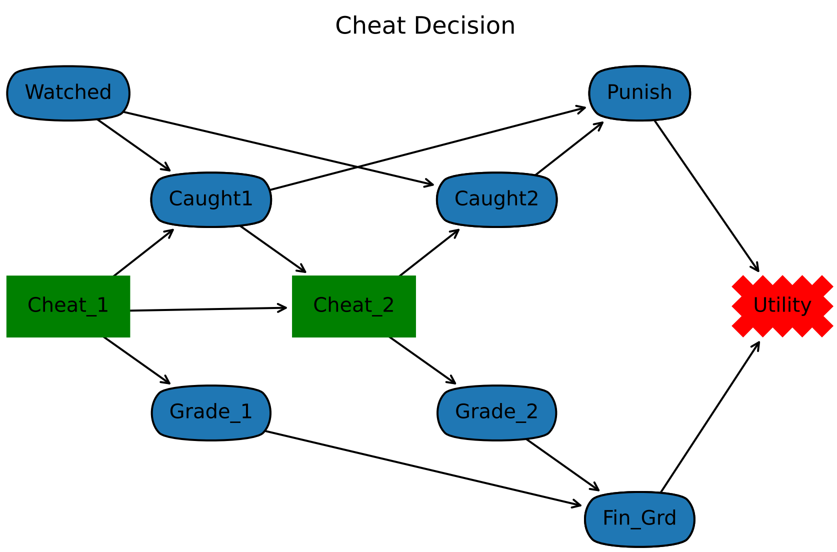


Figure 9.3: Cheating Decision Network

```
              'C':{'A':0.25, 'B':0.5, 'C': 0.25, 'D':0.0},
              'D':{'A':0.25, 'B':0.25, 'C': 0.25, 'D':0.25}},
         'B':{'A':{'A':0.5, 'B':0.5, 'C': 0.0, 'D':0.0},
              'B': {'A':0.0, 'B':1, 'C': 0.0, 'D':0.0},
              'C':{'A':0.0, 'B':0.5, 'C': 0.5, 'D':0.0},
              'D':{'A':0.0, 'B':0.25, 'C': 0.5, 'D':0.25}},
         'C':{'A':{'A':0.25, 'B':0.5, 'C': 0.25, 'D':0.0},
              'B': {'A':0.0, 'B':0.5, 'C': 0.5, 'D':0.0},
              'C':{'A':0.0, 'B':0.0, 'C': 1, 'D':0.0},
              'D':{'A':0.0, 'B':0.0, 'C': 0.5, 'D':0.5}},
         'D':{'A':{'A':0.25, 'B':0.25, 'C': 0.25, 'D':0.25},
              'B': {'A':0.0, 'B':0.25, 'C': 0.5, 'D':0.25},
              'C':{'A':0.0, 'B':0.0, 'C': 0.5, 'D':0.5},
              'D':{'A':0.0, 'B':0.0, 'C': 0, 'D':1.0}}})

utc = UtilityTable([Punish,Fin_Grd],{'None':{'A':100, 'B':90, 'C': 70, 'D':50},
                                  'Suspension':{'A':40, 'B':20, 'C': 10, 'D':0},
                                  'Recorded':{'A':70, 'B':60, 'C': 40, 'D':20}}, position=(1,0.5))

cheat_dn = DecisionNetwork("Cheat Decision",
                          {Punish,Caught2,Watched,Fin_Grd,Grade_2,Grade_1,Cheat_2,Caught1,Cheat_1},
                          {p_wa, p_cc1, p_cc2, p_pun, p_gr1, p_gr2,p_fg,utc})
```

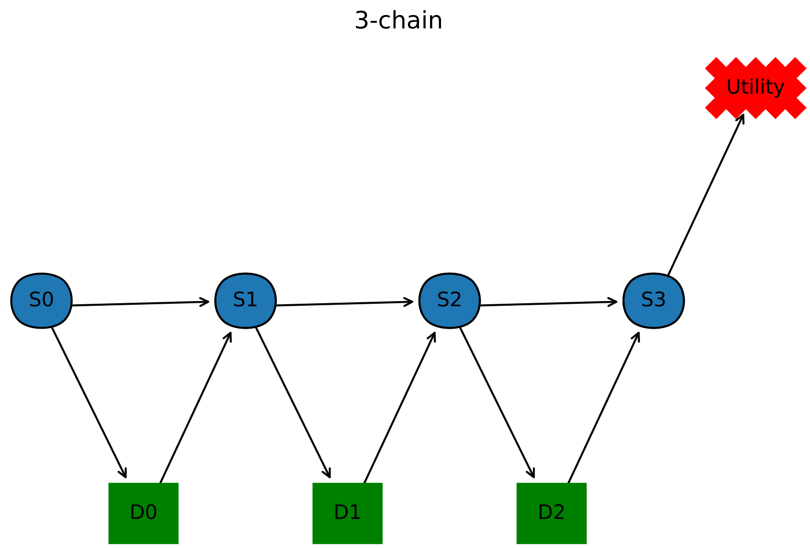


Figure 9.4: A decision network that is a chain of 3 decisions

Chain of 3 decisions

The following example is a finite-stage fully-observable Markov decision process with a single reward (utility) at the end. It is interesting because the parents do not include all predecessors. The methods we use will work without change on this, even though the agent does not condition on all of its previous observations and actions. The output of `ch3.show()` is shown in Figure 9.4.

decnNetworks.py — (continued)

```
S0 = Variable('S0', boolean, position=(0,0.5))
D0 = DecisionVariable('D0', boolean, {S0}, position=(1/7,0.1))
S1 = Variable('S1', boolean, position=(2/7,0.5))
D1 = DecisionVariable('D1', boolean, {S1}, position=(3/7,0.1))
S2 = Variable('S2', boolean, position=(4/7,0.5))
D2 = DecisionVariable('D2', boolean, {S2}, position=(5/7,0.1))
S3 = Variable('S3', boolean, position=(6/7,0.5))

p_s0 = Prob(S0, [], [0.5,0.5])
tr = [[[0.1, 0.9], [0.9, 0.1]], [[0.2, 0.8], [0.8, 0.2]]] # 0 is flip, 1 is keep value
p_s1 = Prob(S1, [D0,S0], tr)
p_s2 = Prob(S2, [D1,S1], tr)
p_s3 = Prob(S3, [D2,S2], tr)

ch3U = UtilityTable([S3],[0,1], position=(7/7,0.9))
```

```
ch3 = DecisionNetwork("3-chain", {S0,D0,S1,D1,S2,D2,S3},{p_s0,p_s1,p_s2,p_s3,ch3U})
#rc3 = RC_DN(ch3)
#rc3.optimize()
#rc3.opt_policy
```

9.1.2 Recursive Conditioning for decision networks

An instance of a `RC_DN` object takes in a decision network. The query method uses recursive conditioning to compute the expected utility of the optimal policy. `self.opt_policy` becomes the optimal policy.

decnNetworks.py — (continued)

```
import math
from probGraphicalModels import GraphicalModel, InferenceMethod
from probFactors import Factor
from utilities import dict_union
from probRC import connected_components

class RC_DN(InferenceMethod):
    """The class that queries graphical models using recursive conditioning

    gm is graphical model to query
    """

    def __init__(self,gm=None):
        self.gm = gm
        self.cache = {(frozenset(), frozenset()):1}
        ## self.max_display_level = 3

    def optimize(self, split_order=None):
        """computes expected utility, and creates optimal decision functions, where
        elim_order is a list of the non-observed non-query variables in gm
        """
        if split_order == None:
            split_order = self.gm.split_order()
        self.opt_policy = {}
        return self.rc({}, self.gm.factors, split_order)
```

The following us the simplest search-based algorithm. It is exponential in the number of variables, so is not very useful. However, it is simple, and useful to understand before looking at the more complicated algorithm. Note that the above code does not call `rc0`; you will need to change the `self.rc` to `self.rc0` in above code to use it.

decnNetworks.py — (continued)

```
    def rc0(self, context, factors, split_order):
        """simplest search algorithm"""
        self.display(2,"calling rc0,",(context,factors),"with SO",split_order)
```

```
        if not factors:
            return 1
        elif to_eval := {fac for fac in factors if fac.can_evaluate(context)}:
            self.display(3,"rc0 evaluating factors",to_eval)
            val = math.prod(fac.get_value(context) for fac in to_eval)
            return val * self.rc0(context, factors-to_eval, split_order)
        else:
            var = split_order[0]
            self.display(3, "rc0 branching on", var)
            if isinstance(var,DecisionVariable):
                assert set(context) <= set(var.parents), f"cannot optimize {var} in context {contex
                maxres = -math.inf
                for val in var.domain:
                    self.display(3,"In rc0, branching on",var,"=",val)
                    newres = self.rc0(dict_union({var:val},context), factors, split_order[1:])
                    if newres > maxres:
                        maxres = newres
                        theval = val
                self.opt_policy[frozenset(context.items())] = (var,theval)
                return maxres
            else:
                total = 0
                for val in var.domain:
                    total += self.rc0(dict_union({var:val},context), factors, split_order[1:])
                self.display(3, "rc0 branching on", var,"returning", total)
                return total
```

We can combine the optimization for decision networks above, with the improvements of recursive conditioning used for graphical models (Section 8.6, page 153).

decnNetworks.py — (continued)

```
    def rc(self, context, factors, split_order):
        """ returns the number \sum_{split_order} \prod_{factors} given assignments in context
        context is a variable:value dictionary
        factors is a set of factors
        split_order is a list of variables in factors that are not in context
        """
        self.display(3,"calling rc,",(context,factors))
        ce = (frozenset(context.items()), frozenset(factors)) # key for the cache entry
        if ce in self.cache:
            self.display(2,"rc cache lookup",(context,factors))
            return self.cache[ce]
#        if not factors: # no factors; needed if you don't have forgetting and caching
#            return 1
        elif vars_not_in_factors := {var for var in context
                                         if not any(var in fac.variables for fac in factors)}:
             # forget variables not in any factor
            self.display(3,"rc forgetting variables", vars_not_in_factors)
            return self.rc({key:val for (key,val) in context.items()
                                if key not in vars_not_in_factors},
```

```
                            factors, split_order)
        elif to_eval := {fac for fac in factors if fac.can_evaluate(context)}:
            # evaluate factors when all variables are assigned
            self.display(3,"rc evaluating factors",to_eval)
            val = math.prod(fac.get_value(context) for fac in to_eval)
            if val == 0:
                return 0
            else:
             return val * self.rc(context, {fac for fac in factors if fac not in to_eval}, split_order)
        elif len(comp := connected_components(context, factors, split_order)) > 1:
            # there are disconnected components
            self.display(2,"splitting into connected components",comp)
            return(math.prod(self.rc(context,f,eo) for (f,eo) in comp))
        else:
            assert split_order, f"split_order empty rc({context},{factors})"
            var = split_order[0]
            self.display(3, "rc branching on", var)
            if isinstance(var,DecisionVariable):
                assert set(context) <= set(var.parents), f"cannot optimize {var} in context {context}"
                maxres = -math.inf
                for val in var.domain:
                    self.display(3,"In rc, branching on",var,"=",val)
                    newres = self.rc(dict_union({var:val},context), factors, split_order[1:])
                    if newres > maxres:
                        maxres = newres
                        theval = val
                self.opt_policy[frozenset(context.items())] = (var,theval)
                self.cache[ce] = maxres
                return maxres
            else:
                total = 0
                for val in var.domain:
                    total += self.rc(dict_union({var:val},context), factors, split_order[1:])
                self.display(3, "rc branching on", var,"returning", total)
                self.cache[ce] = total
                return total
```

Here is how to run the optimize the example decision networks:

decnNetworks.py — (continued)

```
# Umbrella decision network
#urc = RC_DN(umberella_dn)
#urc.optimize()
#urc.opt_policy

#rc_fire = RC_DN(fire_dn)
#rc_fire.optimize()
#rc_fire.opt_policy

#rc_cheat = RC_DN(cheat_dn)
#rc_cheat.optimize()
```

```
#rc_cheat.opt_policy

#rc_ch3 = RC_DN(ch3)
#rc_ch3.optimize()
#rc_ch3.opt_policy
```

9.1.3 Variable elimination for decision networks

VE_DN is variable elimination for decision networks. The method *optimize* is used to optimize all the decisions. Note that *optimize* requires a legal elimination ordering of the random and decision variables, otherwise it will give an exception. (A decision node can only be maximized if the variables that are not its parents have already been eliminated.)

decnNetworks.py — (continued)

```
from probVE import VE

class VE_DN(VE):
    """Variable Elimination for Decision Networks"""
    def __init__(self,dn=None):
        """dn is a decision network"""
        VE.__init__(self,dn)
        self.dn = dn

    def optimize(self,elim_order=None,obs={}):
        if elim_order == None:
                elim_order = reversed(self.gm.split_order())
        policy = []
        proj_factors = [self.project_observations(fact,obs)
                            for fact in self.dn.factors]
        for v in elim_order:
            if isinstance(v,DecisionVariable):
                to_max = [fac for fac in proj_factors
                          if v in fac.variables and set(fac.variables) <= v.all_vars]
                assert len(to_max)==1, "illegal variable order "+str(elim_order)+" at "+str(v)
                newFac = FactorMax(v, to_max[0])
                policy.append(newFac.decision_fun)
                proj_factors = [fac for fac in proj_factors if fac is not to_max[0]]+[newFac]
                self.display(2,"maximizing",v,"resulting factor",newFac.brief() )
                self.display(3,newFac)
            else:
                proj_factors = self.eliminate_var(proj_factors, v)
        assert len(proj_factors)==1,"Should there be only one element of proj_factors?"
        value = proj_factors[0].get_value({})
        return value,policy
```

decnNetworks.py — (continued)

```
class FactorMax(Factor):
    """A factor obtained by maximizing a variable in a factor.
```

```
    Also builds a decision_function. This is based on FactorSum.
    """

    def __init__(self, dvar, factor):
        """dvar is a decision variable.
        factor is a factor that contains dvar and only parents of dvar
        """
        self.dvar = dvar
        self.factor = factor
        vars = [v for v in factor.variables if v is not dvar]
        Factor.__init__(self,vars)
        self.values = [None]*self.size
        self.decision_fun = FactorDF(dvar,vars,[None]*self.size)

    def get_value(self,assignment):
        """lazy implementation: if saved, return saved value, else compute it"""
        index = self.assignment_to_index(assignment)
        if self.values[index]:
            return self.values[index]
        else:
            max_val = float("-inf") # -infinity
            new_asst = assignment.copy()
            for elt in self.dvar.domain:
                new_asst[self.dvar] = elt
                fac_val = self.factor.get_value(new_asst)
                if fac_val>max_val:
                    max_val = fac_val
                    best_elt = elt
            self.values[index] = max_val
            self.decision_fun.values[index] = best_elt
            return max_val
```

A decision function is a stored factor.

decnNetworks.py — (continued)

```
class FactorDF(TabFactor):
    """A decision function"""
    def __init__(self,dvar, vars, values):
        TabStored.__init__(self,vars,values)
        self.dvar = dvar
        self.name = str(dvar) # Used in printing
```

Here are some example queries:

decnNetworks.py — (continued)

```
# Example queries:
# v,p = VE_DN(fire_dn).optimize(); print(v)
# for df in p: print(df,"\n")

# VE_DN.max_display_level = 3 # if you want to show lots of detail
# v,p = VE_DN(cheat_dn).optimize(); print(v)
# 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 recursive conditioning or variable elimination code, as we did for decision networks.

mdpProblem.py — Representations for Markov Decision Processes

```
from utilities import argmaxd
import random
import matplotlib.pyplot as plt
from matplotlib.widgets import Button, CheckButtons

class MDP(object):
    """A Markov Decision Process. Must define:
    self.states the set (or list) of states
    self.actions the set (or list) of actions
    self.discount a real-valued discount
    """

    def __init__(self, states, actions, discount, init=0):
        self.states = states
        self.actions = actions
        self.discount = discount
        self.initv = self.v = {s:init for s in self.states}
        self.initq = self.q = {s: {a: init for a in self.actions} for s in self.states}

    def P(self,s,a):
        """Transition probability function
        returns a dictionary of {s1:p1} such that P(s1 | s,a)=p1. Other probabilities are zero.
        """
        raise NotImplementedError("P") # abstract method

    def R(self,s,a):
        """Reward function R(s,a)
        returns the expected reward for doing a in state s.
        """
        raise NotImplementedError("R") # abstract method
```

Two state partying example (Example 9.27 in Poole and Mackworth [2017]):

mdpExamples.py — MDP Examples

```
from mdpProblem import MDP, GridMDP

class party(MDP):
    """Simple 2-state, 2-Action Partying MDP Example"""
    def __init__(self, discount=0.9):
        states = {'healthy','sick'}
        actions = {'relax', 'party'}
        MDP.__init__(self, states, actions, discount)

    def R(self,s,a):
```

```
        "R(s,a)"
        return { 'healthy': {'relax': 7, 'party': 10},
                 'sick':   {'relax': 0, 'party': 2 }}[s][a]

    def P(self,s,a):
        "returns a dictionary of {s1:p1} such that P(s1 | s,a)=p1. Other probabilities are zero."
        phealthy = { # P('healthy' | s, a)
                    'healthy': {'relax': 0.95, 'party': 0.7},
                    'sick': {'relax': 0.5, 'party': 0.1 }}[s][a]
        return {'healthy':phealthy, 'sick':1-phealthy}
```

The next example is the tiny game from Example 12.1 and Figure 12.1 of Poole and Mackworth [2017]. The state is represented as (x, y) where x counts from zero from the left, and y counts from zero upwards, so the state $(0, 0)$ is on the bottom-left state. The actions are *upC* for up-careful, and *upR* for up-risky. (Note that `GridMDP` is just a type of MDP for which we have methods to show; you can assume it is just `MDP` here).

mdpExamples.py — (continued)

```
class tiny(GridMDP):
    def __init__(self, discount=0.9):
        actions = ['right', 'upC', 'left', 'upR']
        self.x_dim = 2 # x-dimension
        self.y_dim = 3
        states = [(x,y) for x in range(self.x_dim) for y in range(self.y_dim)]
        # for GridMDP
        self.xoff = {'right':0.25, 'upC':0, 'left':-0.25, 'upR':0}
        self.yoff = {'right':0, 'upC':-0.25, 'left':0, 'upR':0.25}
        GridMDP.__init__(self, states, actions, discount)

    def P(self,s,a):
        """return a dictionary of {s1:p1} if P(s1 | s,a)=p1. Other probabilities are zero.
        """
        (x,y) = s
        if a == 'right':
            return {(1,y):1}
        elif a == 'upC':
            return {(x,min(y+1,2)):1}
        elif a == 'left':
            if (x,y) == (0,2): return {(0,0):1}
            else: return {(0,y): 1}
        elif a == 'upR':
            if x==0:
                if y<2: return {(x,y):0.1, (x+1,y):0.1, (x,y+1):0.8}
                else: # at (0,2)
                    return {(0,0):0.1, (1,2): 0.1, (0,2): 0.8}
            elif y < 2: # x==1
                return {(0,y):0.1, (1,y):0.1, (1,y+1):0.8}
            else: # at (1,2)
               return {(0,2):0.1, (1,2): 0.9}

```

```
    def R(self,s,a):
        (x,y) = s
        if a == 'right':
            return [0,-1][x]
        elif a == 'upC':
            return [-1,-1,-2][y]
        elif a == 'left':
            if x==0:
                return [-1, -100, 10][y]
            else: return 0
        elif a == 'upR':
            return [[-0.1, -10, 0.2],[-0.1, -0.1, -0.9]][x][y]
                # at (0,2) reward is 0.1*10+0.8*-1=0.2
```

Here is the domain of Example 9.28 of Poole and Mackworth [2017]. Here the state is represented as (x, y) where x counts from zero from the left, and y counts from zero upwards, so the state $(0, 0)$ is on the bottom-left state.

mdpExamples.py — (continued)

```
class grid(GridMDP):
    """ x_dim * y_dim grid with rewarding states"""
    def __init__(self, discount= 0.9, x_dim=10, y_dim=10):
        self.x_dim = x_dim # size in x-direction
        self.y_dim = y_dim # size in y-direction
        actions = ['up', 'down', 'right', 'left']
        states = [(x,y) for x in range(y_dim) for y in range(y_dim)]
        self.rewarding_states = {(3,2):-10, (3,5):-5, (8,2):10, (7,7):3}
        self.fling_states = {(8,2), (7,7)}
        self.xoff = {'right':0.25, 'up':0, 'left':-0.25, 'down':0}
        self.yoff = {'right':0, 'up':0.25, 'left':0, 'down':-0.25}
        GridMDP.__init__(self, states, actions, discount)

    def intended_next(self,s,a):
        """returns the next state in the direction a.
        This is where the agent will end up if to goes in its intended_direction
             (which it does with probability 0.7).
        """
        (x,y) = s
        if a=='up':
            return (x, y+1 if y+1 < self.y_dim else y)
        if a=='down':
            return (x, y-1 if y > 0 else y)
        if a=='right':
            return (x+1 if x+1 < self.x_dim else x,y)
        if a=='left':
            return (x-1 if x > 0 else x,y)

    def P(self,s,a):
        """return a dictionary of {s1:p1} if P(s1 | s,a)=p1. Other probabilities are zero.
        Corners are tricky because different actions result in same state.
        """
```

```
        if s in self.fling_states:
            return {(0,0): 0.25, (self.x_dim-1,0):0.25, (0,self.y_dim-1):0.25, (self.x_dim-1,self.y_di
        res = dict()
        for ai in self.actions:
            s1 = self.intended_next(s,ai)
            ps1 = 0.7 if ai==a else 0.1
            if s1 in res: # occurs in corners
                res[s1] += ps1
            else:
                res[s1] = ps1
        return res

    def R(self,s,a):
         if s in self.rewarding_states:
             return self.rewarding_states[s]
         else:
             (x,y) = s
             rew = 0
             # rewards from crashing:
             if y==0: ## on bottom.
                 rew += -0.7 if a == 'down' else -0.1
             if y==self.y_dim-1: ## on top.
                 rew += -0.7 if a == 'up' else -0.1
             if x==0: ## on left
                 rew += -0.7 if a == 'left' else -0.1
             if x==self.x_dim-1: ## on right.
                 rew += -0.7 if a == 'right' else -0.1
             return rew
```

9.2.1 Value Iteration

This implements value iteration.

This uses indexes of the states and actions (not the names). The value function is represented so $v[s]$ is the value of state with index s. A Q function is represented so $q[s][a]$ is the value for doing action with index a state with index s. Similarly a policy π 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)

```
    def vi(self, n):
        """carries out n iterations of value iteration, updating value function self.v
        Returns a Q-function, value function, policy
        """
        print("calling vi")
        assert n>0,"You must carry out at least one iteration of vi. n="+str(n)
        #v = v0 if v0 is not None else {s:0 for s in self.states}
        for i in range(n):
            self.q = {s: {a: self.R(s,a)+self.discount*sum(p1*self.v[s1]
                                                          for (s1,p1) in self.P(s,a).items())
```

```
                    for a in self.actions}
                for s in self.states}
            self.v = {s: max(self.q[s][a] for a in self.actions)
                     for s in self.states}
        self.pi = {s: argmaxd(self.q[s])
                     for s in self.states}
        return self.q, self.v, self.pi
```

The following shows how this can be used.

mdpExamples.py — (continued)

```
## Testing value iteration
# Try the following:
# pt = party(discount=0.9)
# pt.vi(1)
# pt.vi(100)
# party(discount=0.99).vi(100)
# party(discount=0.4).vi(100)

# gr = grid()
# gr.show()
# q,v,pi = gr.vi(100)
# q[(7,2)]
```

9.2.2 Showing Grid MDPs

A GridMDP is a type of MDP where we the states are (x,y) positions. It is a special sort of MDP only because we have methods to show it.

mdpProblem.py — (continued)

```
class GridMDP(MDP):
    def __init__(self, states, actions, discount):
        MDP.__init__(self, states, actions, discount)

    def show(self):
        #plt.ion()  # interactive
        fig,(self.ax) = plt.subplots()
        plt.subplots_adjust(bottom=0.2)
        stepB = Button(plt.axes([0.8,0.05,0.1,0.075]), "step")
        stepB.on_clicked(self.on_step)
        resetB = Button(plt.axes([0.6,0.05,0.1,0.075]), "reset")
        resetB.on_clicked(self.on_reset)
        self.qcheck = CheckButtons(plt.axes([0.2,0.05,0.35,0.075]),
                                   ["show q-values","show policy"])
        self.qcheck.on_clicked(self.show_vals)
        self.show_vals(None)
        plt.show()

    def show_vals(self,event):
        self.ax.cla()
```

```
        array = [[self.v[(x,y)] for x in range(self.x_dim)]
                                         for y in range(self.y_dim)]
        self.ax.pcolormesh([x-0.5 for x in range(self.x_dim+1)],
                           [x-0.5 for x in range(self.y_dim+1)],
                           array, edgecolors='black',cmap='summer')
            # for cmap see https://matplotlib.org/stable/tutorials/colors/colormaps.html
        if self.qcheck.get_status()[1]: # "show policy"
                for (x,y) in self.q:
                   maxv = max(self.q[(x,y)][a] for a in self.actions)
                   for a in self.actions:
                       if self.q[(x,y)][a] == maxv:
                          # draw arrow in appropriate direction
                          self.ax.arrow(x,y,self.xoff[a]*2,self.yoff[a]*2,
                                    color='red',width=0.05, head_width=0.2, length_includes_head=True)
        if self.qcheck.get_status()[0]: # "show q-values"
           self.show_q(event)
        else:
           self.show_v(event)
        self.ax.set_xticks(range(self.x_dim))
        self.ax.set_xticklabels(range(self.x_dim))
        self.ax.set_yticks(range(self.y_dim))
        self.ax.set_yticklabels(range(self.y_dim))
        plt.draw()

    def on_step(self,event):
        self.vi(1)
        self.show_vals(event)

    def show_v(self,event):
        """show values"""
        for (x,y) in self.v:
            self.ax.text(x,y,"{val:.2f}".format(val=self.v[(x,y)]),ha='center')

    def show_q(self,event):
        """show q-values"""
        for (x,y) in self.q:
            for a in self.actions:
                self.ax.text(x+self.xoff[a],y+self.yoff[a],
                               "{val:.2f}".format(val=self.q[(x,y)][a]),ha='center')

    def on_reset(self,event):
       self.v = self.initv
       self.q = self.initq
       self.show_vals(event)
```

Figure 9.6 shows the user interface, which can be obtained using `tiny().show()`, resizing it, checking "show q-values" and "show policy", and clicking "step" a few times.

Figure **??** shows the user interface, which can be obtained using `grid().show()`, resizing it, checking "show q-values" and "show policy", and clicking "step" a

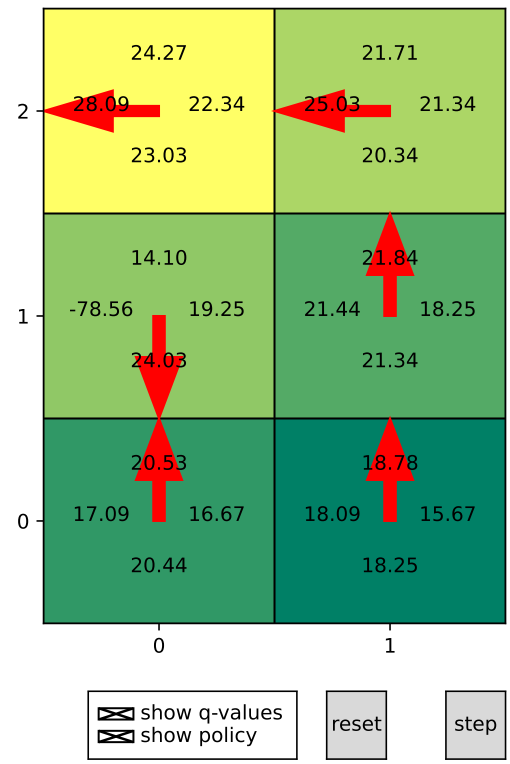


Figure 9.5: Interface for tiny example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The leftmost number is the for the left action; the rightmost number is for the right action; the upper most is for the *upR* (up-risky) action and the lowest number is for the *upC* action. The arrow points to the action(s) with the maximum Q-value.

few times.

Exercise 9.1 Computing q before v may seem like a waste of space because we don't need to store q in order to compute value function or the policy. Change the algorithm so that it loops through the states and actions once per iteration, and only stores the value function and the policy. Note that to get the same results as before, you would need to make sure that you use the previous value of v in the computation not the current value of v. Does using the current value of v hurt the algorithm or make it better (in approaching the actual value function)?

9.2.3 Asynchronous Value Iteration

This implements asynchronous value iteration, storing Q.

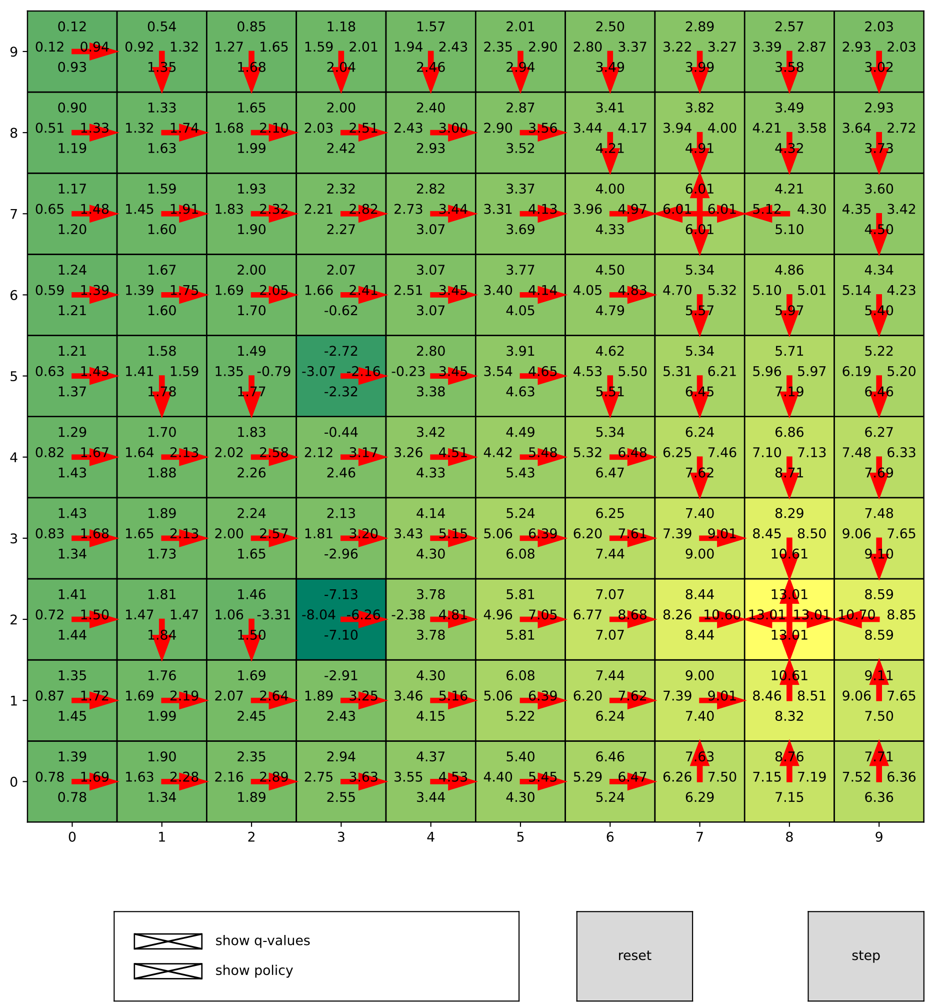


Figure 9.6: Interface for grid example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The leftmost number is the for the left action; the rightmost number is for the right action; the upper most is for the up action and the lowest number is for the down action. The arrow points to the action(s) with the maximum Q-value.

A Q function is represented so $q[s][a]$ is the value for doing action with index a state with index s.

mdpProblem.py — (continued)

```
    def avi(self,n):
          states = list(self.states)
          actions = list(self.actions)
          for i in range(n):
              s = random.choice(states)
              a = random.choice(actions)
              self.q[s][a] = (self.R(s,a) + self.discount *
                                  sum(p1 * max(self.q[s1][a1]
                                                  for a1 in self.actions)
                                        for (s1,p1) in self.P(s,a).items()))
          return Q
```

The following shows how avi can be used.

mdpExamples.py — (continued)

```
## Testing asynchronous value iteration
# Try the following:
# pt = party(discount=0.9)
# pt.avi(10)
# pt.vi(1000)

# gr = grid()
# q = gr.avi(100000)
# q[(7,2)]
```

Exercise 9.2 Implement value iteration that stores the V-values rather than the Q-values. Does it work better than storing Q? (What might better mean?)

Exercise 9.3 In asynchronous value iteration, try a number of different ways to choose the states and actions to update (e.g., sweeping through the state-action pairs, choosing them at random). Note that the best way may be to determine which states have had their Q-values 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.

Chapter 10

Learning with Uncertainty

10.1 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 ith feature in class i is

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

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

```
from learnProblem import Data_set, Learner, Data_from_file
import random
import matplotlib.pyplot as plt

class K_means_learner(Learner):
    def __init__(self,dataset, num_classes):
        self.dataset = dataset
        self.num_classes = num_classes
        self.random_initialize()

    def random_initialize(self):
```

```
        # class_counts[c] is the number of examples with class=c
        self.class_counts = [0]*self.num_classes
        # feature_sum[i][c] is the sum of the values of feature i for class c
        self.feature_sum = [[0]*self.num_classes
                              for feat in self.dataset.input_features]
        for eg in self.dataset.train:
            cl = random.randrange(self.num_classes) # assign eg to random class
            self.class_counts[cl] += 1
            for (ind,feat) in enumerate(self.dataset.input_features):
                self.feature_sum[ind][cl] += feat(eg)
        self.num_iterations = 0
        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 fratures, of the sum-of-squares differences of the class mean and the example value.

learnKMeans.py — (continued)

```
    def distance(self,cl,eg):
        """distance of the eg from the mean of the class"""
        return sum( (self.class_prediction(ind,cl)-feat(eg))**2
                         for (ind,feat) in enumerate(self.dataset.input_features))

    def class_prediction(self,feat_ind,cl):
        """prediction of the class cl on the feature with index feat_ind"""
        if self.class_counts[cl] == 0:
            return 0 # there are no examples so we can choose any value
        else:
            return self.feature_sum[feat_ind][cl]/self.class_counts[cl]

    def class_of_eg(self,eg):
        """class to which eg is assigned"""
        return (min((self.distance(cl,eg),cl)
                        for cl in range(self.num_classes)))[1]
                 # 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)

```
    def k_means_step(self):
        """Updates the model with one step of k-means.
        Returns whether the assignment is stable.
        """
        new_class_counts = [0]*self.num_classes
        # feature_sum[i][c] is the sum of the values of feature i for class c
        new_feature_sum = [[0]*self.num_classes
                              for feat in self.dataset.input_features]
```

```
        for eg in self.dataset.train:
            cl = self.class_of_eg(eg)
            new_class_counts[cl] += 1
            for (ind,feat) in enumerate(self.dataset.input_features):
                new_feature_sum[ind][cl] += feat(eg)
        stable = (new_class_counts == self.class_counts) and (self.feature_sum == new_feature_sum)
        self.class_counts = new_class_counts
        self.feature_sum = new_feature_sum
        self.num_iterations += 1
        return stable


    def learn(self,n=100):
        """do n steps of k-means, or until convergence"""
        i=0
        stable = False
        while i<n and not stable:
            stable = self.k_means_step()
            i += 1
            self.display(1,"Iteration",self.num_iterations,
                            "class counts: ",self.class_counts," Stable=",stable)
        return stable

    def show_classes(self):
        """sorts the data by the class and prints in order.
        For visualizing small data sets
        """
        class_examples = [[] for i in range(self.num_classes)]
        for eg in self.dataset.train:
            class_examples[self.class_of_eg(eg)].append(eg)
        print("Class","Example",sep='\t')
        for cl in range(self.num_classes):
            for eg in class_examples[cl]:
                print(cl,*eg,sep='\t')

    def plot_error(self, maxstep=20):
        """Plots the sum-of-suares error as a function of the number of steps"""
        plt.ion()
        plt.xlabel("step")
        plt.ylabel("Ave sum-of-squares error")
        train_errors = []
        if self.dataset.test:
            test_errors = []
        for i in range(maxstep):
            self.learn(1)
            train_errors.append( sum(self.distance(self.class_of_eg(eg),eg)
                                     for eg in self.dataset.train)
                                 /len(self.dataset.train))
            if self.dataset.test:
                test_errors.append( sum(self.distance(self.class_of_eg(eg),eg)
```

```
                                       for eg in self.dataset.test)
                                 /len(self.dataset.test))
        plt.plot(range(1,maxstep+1),train_errors,
                label=str(self.num_classes)+" classes. Training set")
        if self.dataset.test:
            plt.plot(range(1,maxstep+1),test_errors,
                    label=str(self.num_classes)+" classes. Test set")
        plt.legend()
        plt.draw()

%data = Data_from_file('data/emdata1.csv', num_train=10, target_index=2000) % trivial example
data = Data_from_file('data/emdata2.csv', num_train=10, target_index=2000)
%data = Data_from_file('data/emdata0.csv', num_train=14, target_index=2000) % example from textbook
kml = K_means_learner(data,2)
num_iter=4
print("Class assignment after",num_iter,"iterations:")
kml.learn(num_iter); kml.show_classes()

# Plot the error
# km2=K_means_learner(data,2); km2.plot_error(20) # 2 classes
# km3=K_means_learner(data,3); km3.plot_error(20) # 3 classes
# km13=K_means_learner(data,13); km13.plot_error(20) # 13 classes

# data = Data_from_file('data/carbool.csv', target_index=2000,boolean_features=True)
# kml = K_means_learner(data,3)
# kml.learn(20); kml.show_classes()
# km3=K_means_learner(data,3); km3.plot_error(20) # 3 classes
# km3=K_means_learner(data,30); km3.plot_error(20) # 30 classes
```

Exercise 10.1 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 10.2 If there are many classes, some of the classes can become empty (e.g., try 100 classes with carbool.csv). Implement a way to put some examples into a class, if possible. Two ideas are:

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

(b) In *class_prediction*, we test whether the code is empty, and make a prediction of 0 for an empty class. It is possible to make a different prediction to "steal" an example (but you should make sure that a class has a consistent value for each feature in a loop).

Make your own suggestions, and compare it with the original, and whichever of these you think may work better.

10.2 EM

In the following definition, a class, c, is a integer in range $[0, num\_classes)$. i is an index of a feature, so $feat[i]$ is the ith feature, and a feature is a function from tuples to values. val is a value of a feature.

A model consists of 2 lists, which form the sufficient statistics:

- $class\_counts$ is a list such that $class\_counts[c]$ is the number of tuples with $class = c$, where each tuple is weighted by its probability, i.e.,

$$class\_counts[c] = \sum_{t:class(t)=c} P(t)$$

- $feature\_counts$ is a list such that $feature\_counts[i][val][c]$ is the weighted count of the number of tuples t with $feat[i](t) = val$ and $class(t) = c$, each tuple is weighted by its probability, i.e.,

$$feature\_counts[i][val][c] = \sum_{t:feat[i](t)=val \text{ and } class(t)=c} P(t)$$

learnEM.py — EM Learning

```
from learnProblem import Data_set, Learner, Data_from_file
import random
import math
import matplotlib.pyplot as plt

class EM_learner(Learner):
    def __init__(self,dataset, num_classes):
        self.dataset = dataset
        self.num_classes = num_classes
        self.class_counts = None
        self.feature_counts = None
```

The function *em_step* goes though the training examples, and updates these counts. The first time it is run, when there is no model, it uses random distributions.

learnEM.py — (continued)

```
    def em_step(self, orig_class_counts, orig_feature_counts):
        """updates the model."""
        class_counts = [0]*self.num_classes
        feature_counts = [{val:[0]*self.num_classes
                              for val in feat.frange}
                              for feat in self.dataset.input_features]
        for tple in self.dataset.train:
            if orig_class_counts: # a model exists
                tpl_class_dist = self.prob(tple, orig_class_counts, orig_feature_counts)
            else:                # initially, with no model, return a random distribution
```

```
                tpl_class_dist = random_dist(self.num_classes)
            for cl in range(self.num_classes):
                class_counts[cl] += tpl_class_dist[cl]
                for (ind,feat) in enumerate(self.dataset.input_features):
                    feature_counts[ind][feat(tple)][cl] += tpl_class_dist[cl]
        return class_counts, feature_counts
```

prob computes the probability of a class c for a tuple *tpl*, given the current statistics.

$$\begin{aligned} P(c \mid tple) &\propto P(c) * \prod_i P(X_i{=}tple(i) \mid c) \\ &= \frac{class\_counts[c]}{len(self.dataset)} * \prod_i \frac{feature\_counts[i][feat_i(tple)][c]}{class\_counts[c]} \\ &\propto \frac{\prod_i feature\_counts[i][feat_i(tple)][c]}{class\_counts[c]^{|feats|-1}} \end{aligned}$$

The last step is because $len(self.dataset)$ is a constant (independent of c). $class\_counts[c]$ can be taken out of the product, but needs to be raised to the power of the number of features, and one of them cancels.

learnEM.py — (continued)

```
    def prob(self, tple, class_counts, feature_counts):
        """returns a distribution over the classes for tuple tple in the model defined by the counts
        """
        feats = self.dataset.input_features
        unnorm = [prod(feature_counts[i][feat(tple)][c]
                        for (i,feat) in enumerate(feats))
                      /(class_counts[c]**(len(feats)-1))
                    for c in range(self.num_classes)]
        thesum = sum(unnorm)
        return [un/thesum for un in unnorm]
```

learn does n steps of EM:

learnEM.py — (continued)

```
    def learn(self,n):
        """do n steps of em"""
        for i in range(n):
            self.class_counts,self.feature_counts = self.em_step(self.class_counts,
                                                                 self.feature_counts)
```

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

learnEM.py — (continued)

```
    def show_class(self,c):
        """sorts the data by the class and prints in order.
        For visualizing small data sets
        """
        sorted_data = sorted((self.prob(tpl,self.class_counts,self.feature_counts)[c],
```

```
                            ind,  # preserve ordering for equal probabilities
                            tpl)
                           for (ind,tpl) in enumerate(self.dataset.train))
        for cc,r,tpl in sorted_data:
            print(cc,*tpl,sep='\t')
```

The following are for evaluating the classes.

The probability of a tuple can be evaluated by marginalizing over the classes:

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

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

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

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

learnEM.py — (continued)

```
    def logloss(self,tple):
        """returns the logloss of the prediction on tple, which is -log(P(tple))
        based on the current class counts and feature counts
        """
        feats = self.dataset.input_features
        res = 0
        cc = self.class_counts
        fc = self.feature_counts
        for c in range(self.num_classes):
            res += prod(fc[i][feat(tple)][c]
                        for (i,feat) in enumerate(feats))/(cc[c]**(len(feats)-1))
        if res>0:
            return -math.log2(res/len(self.dataset.train))
        else:
            return float("inf") #infinity

    def plot_error(self, maxstep=20):
        """Plots the logloss error as a function of the number of steps"""
        plt.ion()
        plt.xlabel("step")
        plt.ylabel("Ave Logloss (bits)")
        train_errors = []
        if self.dataset.test:
            test_errors = []
        for i in range(maxstep):
            self.learn(1)
```

```
            train_errors.append( sum(self.logloss(tple) for tple in self.dataset.train)
                                 /len(self.dataset.train))
            if self.dataset.test:
                test_errors.append( sum(self.logloss(tple) for tple in self.dataset.test)
                                     /len(self.dataset.test))
        plt.plot(range(1,maxstep+1),train_errors,
                 label=str(self.num_classes)+" classes. Training set")
        if self.dataset.test:
            plt.plot(range(1,maxstep+1),test_errors,
                     label=str(self.num_classes)+" classes. Test set")
        plt.legend()
        plt.draw()

def prod(L):
    """returns the product of the elements of L"""
    res = 1
    for e in L:
        res *= e
    return res

def random_dist(k):
    """generate k random numbers that sum to 1"""
    res = [random.random() for i in range(k)]
    s = sum(res)
    return [v/s for v in res]

data = Data_from_file('data/emdata2.csv', num_train=10, target_index=2000)
eml = EM_learner(data,2)
num_iter=2
print("Class assignment after",num_iter,"iterations:")
eml.learn(num_iter); eml.show_class(0)

# Plot the error
# em2=EM_learner(data,2); em2.plot_error(40) # 2 classes
# em3=EM_learner(data,3); em3.plot_error(40) # 3 classes
# em13=EM_learner(data,13); em13.plot_error(40) # 13 classes

# data = Data_from_file('data/carbool.csv', target_index=2000,boolean_features=False)
# [f.frange for f in data.input_features]
# eml = EM_learner(data,3)
# eml.learn(20); eml.show_class(0)
# em3=EM_learner(data,3); em3.plot_error(60) # 3 classes
# em3=EM_learner(data,30); em3.plot_error(60) # 30 classes
```

Exercise 10.3 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 10.4 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.)

Chapter 11

Multiagent Systems

11.1 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.

11.1.1 Creating a two-player game

masProblem.py — A Multiagent Problem

```
from display import Displayable

class Node(Displayable):
    """A node in a search tree. It has a
    name a string
    isMax is True if it is a maximizing node, otherwise it is minimizing node
    children is the list of children
    value is what it evaluates to if it is a leaf.
    """
    def __init__(self, name, isMax, value, children):
        self.name = name
        self.isMax = isMax
        self.value = value
        self.allchildren = children

    def isLeaf(self):
        """returns true of this is a leaf node"""
        return self.allchildren is None

```

```
    def children(self):
        """returns the list of all children."""
        return self.allchildren

    def evaluate(self):
        """returns the evaluation for this node if it is a leaf"""
        return self.value
```

The following gives the tree from Figure 11.5 of the book. Note how 888 is used as a value here, but never appears in the trace.

masProblem.py — (continued)

```
fig10_5 = Node("a",True,None, [
            Node("b",False,None, [
                Node("d",True,None, [
                    Node("h",False,None, [
                        Node("h1",True,7,None),
                        Node("h2",True,9,None)]),
                    Node("i",False,None, [
                        Node("i1",True,6,None),
                        Node("i2",True,888,None)])]),
                Node("e",True,None, [
                    Node("j",False,None, [
                        Node("j1",True,11,None),
                        Node("j2",True,12,None)]),
                    Node("k",False,None, [
                        Node("k1",True,888,None),
                        Node("k2",True,888,None)])])]),
            Node("c",False,None, [
                Node("f",True,None, [
                    Node("l",False,None, [
                        Node("l1",True,5,None),
                        Node("l2",True,888,None)]),
                    Node("m",False,None, [
                        Node("m1",True,4,None),
                        Node("m2",True,888,None)])]),
                Node("g",True,None, [
                    Node("n",False,None, [
                        Node("n1",True,888,None),
                        Node("n2",True,888,None)]),
                    Node("o",False,None, [
                        Node("o1",True,888,None),
                        Node("o2",True,888,None)])])])])
```

The following is a representation of a **magic-sum game**, where players take turns picking a number in the range $[1, 9]$, and the first player to have 3 numbers that sum to 15 wins. Note that this is a syntactic variant of **tic-tac-toe** or **naughts and crosses**. To see this, consider the numbers on a **magic square** (Figure 11.1); 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 11.1: Magic Square

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

masProblem.py — (continued)

```
class Magic_sum(Node):
    def __init__(self, xmove=True, last_move=None,
                 available=[1,2,3,4,5,6,7,8,9], x=[], o=[]):
        """This is a node in the search for the magic-sum game.
        xmove is True if the next move belongs to X.
        last_move is the number selected in the last move
        available is the list of numbers that are available to be chosen
        x is the list of numbers already chosen by x
        o is the list of numbers already chosen by o
        """
        self.isMax = self.xmove = xmove
        self.last_move = last_move
        self.available = available
        self.x = x
        self.o = o
        self.allchildren = None #computed on demand
        lm = str(last_move)
        self.name = "start" if not last_move else "o="+lm if xmove else "x="+lm

    def children(self):
        if self.allchildren is None:
            if self.xmove:
                self.allchildren = [
                    Magic_sum(xmove = not self.xmove,
                              last_move = sel,
                              available = [e for e in self.available if e is not sel],
                              x = self.x+[sel],
                              o = self.o)
                            for sel in self.available]
            else:
                self.allchildren = [
                    Magic_sum(xmove = not self.xmove,
                              last_move = sel,
                              available = [e for e in self.available if e is not sel],
                              x = self.x,
                              o = self.o+[sel])
                            for sel in self.available]
        return self.allchildren

```

```
    def isLeaf(self):
        """A leaf has no numbers available or is a win for one of the players.
        We only need to check for a win for o if it is currently x's turn,
        and only check for a win for x if it is o's turn (otherwise it would
        have been a win earlier).
        """
        return (self.available == [] or
                (sum_to_15(self.last_move,self.o)
                 if self.xmove
                 else sum_to_15(self.last_move,self.x)))

    def evaluate(self):
        if self.xmove and sum_to_15(self.last_move,self.o):
            return -1
        elif not self.xmove and sum_to_15(self.last_move,self.x):
            return 1
        else:
            return 0

def sum_to_15(last,selected):
    """is true if last, toegether with two other elements of selected sum to 15.
    """
    return any(last+a+b == 15
               for a in selected if a != last
               for b in selected if b != last and b != a)
```

11.1.2 Minimax and α-β Pruning

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

masMiniMax.py — Minimax search with alpha-beta pruning

```
def minimax(node,depth):
    """returns the value of node, and a best path for the agents
    """
    if node.isLeaf():
        return node.evaluate(),None
    elif node.isMax:
        max_score = float("-inf")
        max_path = None
        for C in node.children():
            score,path = minimax(C,depth+1)
            if score > max_score:
                max_score = score
                max_path = C.name,path
        return max_score,max_path
    else:
        min_score = float("inf")
        min_path = None
        for C in node.children():
            score,path = minimax(C,depth+1)
            if score < min_score:
                min_score = score
                min_path = C.name,path
        return min_score,min_path
```

The following is a depth-first minimax with α-β **pruning**. It returns the value for a node as well as a best path for the agents.

masMiniMax.py — (continued)

```
def minimax_alpha_beta(node,alpha,beta,depth=0):
    """node is a Node, alpha and beta are cutoffs, depth is the depth
    returns value, path
    where path is a sequence of nodes that results in the value
    """
    node.display(2,"  "*depth,"minimax_alpha_beta(",node.name,", ",alpha, ", ", beta,")")
    best=None     # only used if it will be pruned
    if node.isLeaf():
        node.display(2,"  "*depth,"returning leaf value",node.evaluate())
        return node.evaluate(),None
    elif node.isMax:
        for C in node.children():
            score,path = minimax_alpha_beta(C,alpha,beta,depth+1)
            if score >= beta:  # beta pruning
                node.display(2,"  "*depth,"pruned due to beta=",beta,"C=",C.name)
                return score, None
            if score > alpha:
                alpha = score
                best = C.name, path
        node.display(2,"  "*depth,"returning max alpha",alpha,"best",best)
        return alpha,best
    else:
        for C in node.children():
            score,path = minimax_alpha_beta(C,alpha,beta,depth+1)
            if score <= alpha:   # alpha pruning
                node.display(2,"  "*depth,"pruned due to alpha=",alpha,"C=",C.name)
                return score, None
            if score < beta:
                beta=score
                best = C.name,path
        node.display(2,"  "*depth,"returning min beta",beta,"best=",best)
        return beta,best
```

Testing:

masMiniMax.py — (continued)

```
from masProblem import fig10_5, Magic_sum, Node

# Node.max_display_level=2 # print detailed trace
# minimax_alpha_beta(fig10_5, -9999, 9999,0)
# minimax_alpha_beta(Magic_sum(), -9999, 9999,0)

#To see how much time alpha-beta pruning can save over minimax, uncomment the following:
## import timeit
## timeit.Timer("minimax(Magic_sum(),0)",setup="from __main__ import minimax, Magic_sum"
##              ).timeit(number=1)
## trace=False
```

```
## timeit.Timer("minimax_alpha_beta(Magic_sum(), -9999, 9999,0)",
##            setup="from __main__ import minimax_alpha_beta, Magic_sum"
##            ).timeit(number=1)
```

Chapter 12

Reinforcement Learning

12.1 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

```
import random
from display import Displayable
from utilities import flip

class RL_env(Displayable):
    def __init__(self,actions,state):
        self.actions = actions # set of actions
        self.state = state     # initial state

    def do(self, action):
        """do action
        returns state,reward
        """
        raise NotImplementedError("RL_env.do") # abstract method
```

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

rlProblem.py — (continued)

```
class Healthy_env(RL_env):
    def __init__(self):
        RL_env.__init__(self,["party","relax"], "healthy")

```

```
    def do(self, action):
        """updates the state based on the agent doing action.
        returns state,reward
        """
        if self.state=="healthy":
            if action=="party":
                self.state = "healthy" if flip(0.7) else "sick"
                reward = 10
            else: # action=="relax"
                self.state = "healthy" if flip(0.95) else "sick"
                reward = 7
        else: # self.state=="sick"
            if action=="party":
                self.state = "healthy" if flip(0.1) else "sick"
                reward = 2
            else:
                self.state = "healthy" if flip(0.5) else "sick"
                reward = 0
        return self.state,reward
```

12.1.1 Simulating an environment from an MDP

Given the definition for an MDP (page 195), *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)

```
class Env_from_MDP(RL_env):
    def __init__(self, mdp):
        initial_state = mdp.states[0]
        RL_env.__init__(self,mdp.actions, initial_state)
        self.mdp = mdp
        self.action_index = {action:index for (index,action) in enumerate(mdp.actions)}
        self.state_index = {state:index for (index,state) in enumerate(mdp.states)}

    def do(self, action):
        """updates the state based on the agent doing action.
        returns state,reward
        """
        action_ind = self.action_index[action]
        state_ind = self.state_index[self.state]
        self.state = pick_from_dist(self.mdp.trans[state_ind][action_ind], self.mdp.states)
        reward = self.mdp.reward[state_ind][action_ind]
        return self.state, reward

def pick_from_dist(dist,values):
```

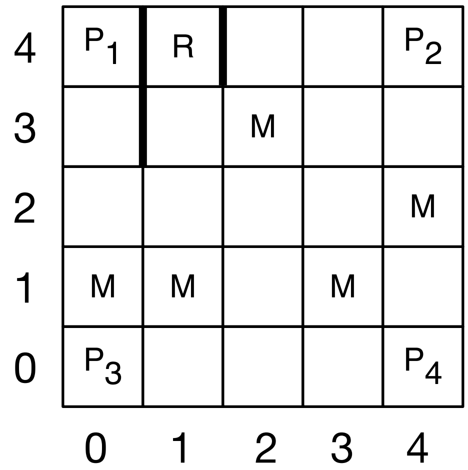


Figure 12.1: Monster game

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

12.1.2 Simple Game

This is for the game depicted in Figure 12.1.

rlSimpleEnv.py — Simple game

```
import random
from utilities import flip
from rlProblem import RL_env

class Simple_game_env(RL_env):
    xdim = 5
    ydim = 5

    vwalls = [(0,3), (0,4), (1,4)] # vertical walls right of these locations
    hwalls = [] # not implemented
    crashed_reward = -1

    prize_locs = [(0,0), (0,4), (4,0), (4,4)]
    prize_apears_prob = 0.3
    prize_reward = 10
```

```

    monster_locs = [(0,1), (1,1), (2,3), (3,1), (4,2)]
    monster_appears_prob = 0.4
    monster_reward_when_damaged = -10
    repair_stations = [(1,4)]

    actions = ["up","down","left","right"]

    def __init__(self):
        # State:
        self.x = 2
        self.y = 2
        self.damaged = False
        self.prize = None
        # Statistics
        self.number_steps = 0
        self.total_reward = 0
        self.min_reward = 0
        self.min_step = 0
        self.zero_crossing = 0
        RL_env.__init__(self, Simple_game_env.actions,
                        (self.x, self.y, self.damaged, self.prize))
        self.display(2,"","Step","Tot Rew","Ave Rew",sep="\t")

    def do(self,action):
        """updates the state based on the agent doing action.
        returns state,reward
        """
        reward = 0.0
        # A prize can appear:
        if self.prize is None and flip(self.prize_apears_prob):
                self.prize = random.choice(self.prize_locs)
        # Actions can be noisy
        if flip(0.4):
            actual_direction = random.choice(self.actions)
        else:
            actual_direction = action
        # Modeling the actions given the actual direction
        if actual_direction == "right":
            if self.x==self.xdim-1 or (self.x,self.y) in self.vwalls:
                reward += self.crashed_reward
            else:
                self.x += 1
        elif actual_direction == "left":
            if self.x==0 or (self.x-1,self.y) in self.vwalls:
                reward += self.crashed_reward
            else:
                self.x += -1
        elif actual_direction == "up":
            if self.y==self.ydim-1:
```

```
                reward += self.crashed_reward
            else:
                self.y += 1
        elif actual_direction == "down":
            if self.y==0:
                reward += self.crashed_reward
            else:
                self.y += -1
        else:
            raise RuntimeError("unknown_direction "+str(direction))

        # Monsters
        if (self.x,self.y) in self.monster_locs and flip(self.monster_appears_prob):
            if self.damaged:
                reward += self.monster_reward_when_damaged
            else:
                self.damaged = True
        if (self.x,self.y) in self.repair_stations:
            self.damaged = False

        # Prizes
        if (self.x,self.y) == self.prize:
            reward += self.prize_reward
            self.prize = None

        # Statistics
        self.number_steps += 1
        self.total_reward += reward
        if self.total_reward < self.min_reward:
            self.min_reward = self.total_reward
            self.min_step = self.number_steps
        if self.total_reward>0 and reward>self.total_reward:
            self.zero_crossing = self.number_steps
        self.display(2,"",self.number_steps,self.total_reward,
                      self.total_reward/self.number_steps,sep="\t")

        return (self.x, self.y, self.damaged, self.prize), reward
```

12.1.3 Evaluation and Plotting

rlPlot.py — RL Plotter

```
import matplotlib.pyplot as plt

def plot_rl(ag, label=None, yplot='Total', step_size=None,
            steps_explore=1000, steps_exploit=1000, xscale='linear'):
    """
    plots the agent ag
    label is the label for the plot
    yplot is 'Average' or 'Total'
```

```
    step_size is the number of steps between each point plotted
    steps_explore is the number of steps the agent spends exploring
    steps_exploit is the number of steps the agent spends exploiting
    xscale is 'log' or 'linear'

    returns total reward when exploring, total reward when exploiting
    """
    assert yplot in ['Average','Total']
    if step_size is None:
       step_size = max(1,(steps_explore+steps_exploit)//500)
    if label is None:
       label = ag.label
    ag.max_display_level,old_mdl = 1,ag.max_display_level
    plt.ion()
    plt.xscale(xscale)
    plt.xlabel("step")
    plt.ylabel(yplot+" reward")
    steps = []      # steps
    rewards = []    # return
    ag.restart()
    step = 0
    while step < steps_explore:
       ag.do(step_size)
       step += step_size
       steps.append(step)
       if yplot == "Average":
          rewards.append(ag.acc_rewards/step)
       else:
          rewards.append(ag.acc_rewards)
    acc_rewards_exploring = ag.acc_rewards
    ag.explore,explore_save = 0,ag.explore
    while step < steps_explore+steps_exploit:
       ag.do(step_size)
       step += step_size
       steps.append(step)
       if yplot == "Average":
          rewards.append(ag.acc_rewards/step)
       else:
          rewards.append(ag.acc_rewards)
    plt.plot(steps,rewards,label=label)
    plt.legend(loc="upper left")
    plt.draw()
    ag.max_display_level = old_mdl
    ag.explore=explore_save
    return acc_rewards_exploring, ag.acc_rewards-acc_rewards_exploring
```

12.2 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

```
import random
from display import Displayable
from utilities import argmaxe, flip

class RL_agent(Displayable):
    """An RL_Agent
    has percepts (s, r) for some state s and real reward r
    """
```

rlQLearner.py — (continued)

```
class Q_learner(RL_agent):
    """A Q-learning agent has
    belief-state consisting of
        state is the previous state
        q is a {(state,action):value} dict
        visits is a {(state,action):n} dict. n is how many times action was done in state
        acc_rewards is the accumulated reward

    it observes (s, r) for some world-state s and real reward r
    """
```

rlQLearner.py — (continued)

```
    def __init__(self, env, discount, explore=0.1, fixed_alpha=True, alpha=0.2,
                 alpha_fun=lambda k:1/k,
                 qinit=0, label="Q_learner"):
        """env is the environment to interact with.
        discount is the discount factor
        explore is the proportion of time the agent will explore
        fixed_alpha specifies whether alpha is fixed or varies with the number of visits
        alpha is the weight of new experiences compared to old experiences
        alpha_fun is a function that computes alpha from the number of visits
        qinit is the initial value of the Q's
        label is the label for plotting
        """
        RL_agent.__init__(self)
        self.env = env
        self.actions = env.actions
        self.discount = discount
        self.explore = explore
        self.fixed_alpha = fixed_alpha
        self.alpha = alpha
```

```
        self.alpha_fun = alpha_fun
        self.qinit = qinit
        self.label = label
        self.restart()
```

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

rlQLearner.py — (continued)

```
    def restart(self):
        """make the agent relearn, and reset the accumulated rewards
        """
        self.acc_rewards = 0
        self.state = self.env.state
        self.q = {}
        self.visits = {}
```

do takes in the number of steps.

rlQLearner.py — (continued)

```
    def do(self,num_steps=100):
        """do num_steps of interaction with the environment"""
        self.display(2,"s\ta\tr\ts'\tQ")
        alpha = self.alpha
        for i in range(num_steps):
            action = self.select_action(self.state)
            next_state,reward = self.env.do(action)
            if not self.fixed_alpha:
                k = self.visits[(self.state, action)] = self.visits.get((self.state, action),0)+1
                alpha = self.alpha_fun(k)
            self.q[(self.state, action)] = (
                (1-alpha) * self.q.get((self.state, action),self.qinit)
                + alpha * (reward + self.discount
                                  * max(self.q.get((next_state, next_act),self.qinit)
                                        for next_act in self.actions)))
            self.display(2,self.state, action, reward, next_state,
                         self.q[(self.state, action)], sep='\t')
            self.state = next_state
            self.acc_rewards += reward
```

select_action us used to select the next action to perform. This can be reimplemented to give a different exploration strategy.

rlQLearner.py — (continued)

```
    def select_action(self, state):
        """returns an action to carry out for the current agent
        given the state, and the q-function
        """
        if flip(self.explore):
            return random.choice(self.actions)
        else:
            return argmaxe((next_act, self.q.get((state, next_act),self.qinit))
```

```
                                        for next_act in self.actions)
```

Exercise 12.1 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 12.2 Implement SARSA. Hint: it does not do a *max* in *do*. Instead it needs to choose *next_act* before it does the update.

12.2.1 Testing Q-learning

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

rlQTest.py — RL Q Tester

```
from rlProblem import Healthy_env
from rlQLearner import Q_learner
from rlPlot import plot_rl

env = Healthy_env()
ag = Q_learner(env, 0.7)
ag_opt = Q_learner(env, 0.7, qinit=100, label="optimistic" ) # optimistic agent
ag_exp_l = Q_learner(env, 0.7, explore=0.01, label="less explore")
ag_exp_m = Q_learner(env, 0.7, explore=0.5, label="more explore")
ag_disc = Q_learner(env, 0.9, qinit=100, label="disc 0.9")
ag_va = Q_learner(env, 0.7, qinit=100,fixed_alpha=False,alpha_fun=lambda k:10/(9+k),label="alpha=1

# ag.max_display_level = 2
# ag.do(20)
# ag.q   # get the learned q-values
# ag.max_display_level = 1
# ag.do(1000)
# ag.q   # get the learned q-values
# plot_rl(ag,yplot="Average")
# plot_rl(ag_opt,yplot="Average")
# plot_rl(ag_exp_l,yplot="Average")
# plot_rl(ag_exp_m,yplot="Average")
# plot_rl(ag_disc,yplot="Average")
# plot_rl(ag_va,yplot="Average")

from mdpExamples import mdpt
from rlProblem import Env_from_MDP
envt = Env_from_MDP(mdpt)
agt = Q_learner(envt, 0.8)
# agt.do(20)

from rlSimpleEnv import Simple_game_env
senv = Simple_game_env()
sag1 = Q_learner(senv,0.9,explore=0.2,fixed_alpha=True,alpha=0.1)
# plot_rl(sag1,steps_explore=100000,steps_exploit=100000,label="alpha="+str(sag1.alpha))
sag2 = Q_learner(senv,0.9,explore=0.2,fixed_alpha=False)
# plot_rl(sag2,steps_explore=100000,steps_exploit=100000,label="alpha=1/k")
```

```
sag3 = Q_learner(senv,0.9,explore=0.2,fixed_alpha=False,alpha_fun=lambda k:10/(9+k))
# plot_rl(sag3,steps_explore=100000,steps_exploit=100000,label="alpha=10/(9+k)")
```

12.3 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.

A model-based reinforcement learner builds a Markov decision process model of the domain, simultaneously learns the model and plans with that model.

The model-based reinforcement learner 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

```
import random
from rlQLearner import RL_agent
from display import Displayable
from utilities import argmaxe, flip

class Model_based_reinforcement_learner(RL_agent):
    """A Model-based reinforcement learner
    """

```

```
    def __init__(self, env, discount, explore=0.1, qinit=0,
                 updates_per_step=10, label="MBR_learner"):
        """env is the environment to interact with.
        discount is the discount factor
        explore is the proportion of time the agent will explore
        qinit is the initial value of the Q's
        updates_per_step is the number of AVI updates per action
        label is the label for plotting
        """
        RL_agent.__init__(self)
        self.env = env
        self.actions = env.actions
        self.discount = discount
        self.explore = explore
        self.qinit = qinit
        self.updates_per_step = updates_per_step
        self.label = label
        self.restart()
```

rlModelLearner.py — (continued)

```
    def restart(self):
        """make the agent relearn, and reset the accumulated rewards
        """
        self.acc_rewards = 0
        self.state = self.env.state
        self.q = {}          # {(st,action):q_value} map
        self.r = {}          # {(st,action):reward} map
        self.t = {}          # {(st,action,st_next):count} map
        self.visits = {}     # {(st,action):count} map
        self.res_states = {}  # {(st,action):set_of_states} map
        self.visits_list = [] # list of (st,action)
        self.previous_action = None
```

rlModelLearner.py — (continued)

```
    def do(self,num_steps=100):
        """do num_steps of interaction with the environment
        for each action, do updates_per_step iterations of asynchronous value iteration
        """
        for step in range(num_steps):
            pst = self.state   # previous state
            action = self.select_action(pst)
            self.state,reward = self.env.do(action)
            self.acc_rewards += reward
            self.t[(pst,action,self.state)] = self.t.get((pst, action,self.state),0)+1
            if (pst,action) in self.visits:
                self.visits[(pst,action)] += 1
                self.r[(pst,action)] += (reward-self.r[(pst,action)])/self.visits[(pst,action)]
                self.res_states[(pst,action)].add(self.state)
            else:
                self.visits[(pst,action)] = 1
```

```
                self.r[(pst,action)] = reward
                self.res_states[(pst,action)] = {self.state}
                self.visits_list.append((pst,action))
            st,act = pst,action   #initial state-action pair for AVI
            for update in range(self.updates_per_step):
                self.q[(st,act)] = self.r[(st,act)]+self.discount*(
                    sum(self.t[st,act,rst]/self.visits[st,act]*
                        max(self.q.get((rst,nact),self.qinit) for nact in self.actions)
                        for rst in self.res_states[(st,act)]))
                st,act = random.choice(self.visits_list)
```

rlModelLearner.py — (continued)

```
    def select_action(self, state):
        """returns an action to carry out for the current agent
        given the state, and the q-function
        """
        if flip(self.explore):
            return random.choice(self.actions)
        else:
            return argmaxe((next_act, self.q.get((state, next_act),self.qinit))
                                  for next_act in self.actions)
```

rlModelLearner.py — (continued)

```
from rlQTest import senv # simple game environment
mbl1 = Model_based_reinforcement_learner(senv,0.9,updates_per_step=10)
# plot_rl(mbl1,steps_explore=100000,steps_exploit=100000,label="model-based(10)")
mbl2 = Model_based_reinforcement_learner(senv,0.9,updates_per_step=1)
# plot_rl(mbl2,steps_explore=100000,steps_exploit=100000,label="model-based(1)")
```

Exercise 12.3 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 12.4 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 12.5 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?

12.4 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.

12.4.1 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

```
from rlSimpleEnv import Simple_game_env
from rlProblem import RL_env

def get_features(state,action):
    """returns the list of feature values for the state-action pair
    """
    assert action in Simple_game_env.actions
    (x,y,d,p) = state
    # f1: would go to a monster
    f1 = monster_ahead(x,y,action)
    # f2: would crash into wall
    f2 = wall_ahead(x,y,action)
    # f3: action is towards a prize
    f3 = towards_prize(x,y,action,p)
    # f4: damaged and action is toward repair station
    f4 = towards_repair(x,y,action) if d else 0
    # f5: damaged and towards monster
    f5 = 1 if d and f1 else 0
    # f6: damaged
    f6 = 1 if d else 0
    # f7: not damaged
    f7 = 1-f6
    # f8: damaged and prize ahead
    f8 = 1 if d and f3 else 0
    # f9: not damaged and prize ahead
    f9 = 1 if not d and f3 else 0
    features = [1,f1,f2,f3,f4,f5,f6,f7,f8,f9]
    for pr in Simple_game_env.prize_locs+[None]:
        if p==pr:
            features += [x, 4-x, y, 4-y]
        else:
            features += [0, 0, 0, 0]
    # fp04 feature for y when prize is at 0,4
    # this knows about the wall to the right of the prize
    if p==(0,4):
        if x==0:
            fp04 = y
        elif y<3:
            fp04 = y
        else:
```

```
            fp04 = 4-y
    else:
        fp04 = 0
    features.append(fp04)
    return features

def monster_ahead(x,y,action):
    """returns 1 if the location expected to get to by doing
    action from (x,y) can contain a monster.
    """
    if action == "right" and (x+1,y) in Simple_game_env.monster_locs:
        return 1
    elif action == "left" and (x-1,y) in Simple_game_env.monster_locs:
        return 1
    elif action == "up" and (x,y+1) in Simple_game_env.monster_locs:
        return 1
    elif action == "down" and (x,y-1) in Simple_game_env.monster_locs:
        return 1
    else:
        return 0

def wall_ahead(x,y,action):
    """returns 1 if there is a wall in the direction of action from (x,y).
    This is complicated by the internal walls.
    """
    if action == "right" and (x==Simple_game_env.xdim-1 or (x,y) in Simple_game_env.vwalls):
        return 1
    elif action == "left" and (x==0 or (x-1,y) in Simple_game_env.vwalls):
        return 1
    elif action == "up" and y==Simple_game_env.ydim-1:
        return 1
    elif action == "down" and y==0:
        return 1
    else:
        return 0

def towards_prize(x,y,action,p):
    """action goes in the direction of the prize from (x,y)"""
    if p is None:
        return 0
    elif p==(0,4): # take into account the wall near the top-left prize
        if action == "left" and (x>1 or x==1 and y<3):
            return 1
        elif action == "down" and (x>0 and y>2):
            return 1
        elif action == "up" and (x==0 or y<2):
            return 1
        else:
            return 0
    else:
```

```
        px,py = p
        if p==(4,4) and x==0:
            if (action=="right" and y<3) or (action=="down" and y>2) or (action=="up" and y<2):
                return 1
            else:
                return 0
        if (action == "up" and y<py) or (action == "down" and py<y):
            return 1
        elif (action == "left" and px<x) or (action == "right" and x<px):
            return 1
        else:
            return 0

def towards_repair(x,y,action):
    """returns 1 if action is towards the repair station.
    """
    if action == "up" and (x>0 and y<4 or x==0 and y<2):
        return 1
    elif action == "left" and x>1:
        return 1
    elif action == "right" and x==0 and y<3:
        return 1
    elif action == "down" and x==0 and y>2:
        return 1
    else:
        return 0

def simp_features(state,action):
    """returns a list of feature values for the state-action pair
    """
    assert action in Simple_game_env.actions
    (x,y,d,p) = state
    # f1: would go to a monster
    f1 = monster_ahead(x,y,action)
    # f2: would crash into wall
    f2 = wall_ahead(x,y,action)
    # f3: action is towards a prize
    f3 = towards_prize(x,y,action,p)
    return [1,f1,f2,f3]
```

12.4.2 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

```
import random
from rlQLearner import RL_agent
from display import Displayable
```

```
from utilities import argmaxe, flip

class SARSA_LFA_learner(RL_agent):
    """A SARSA_LFA learning agent has
    belief-state consisting of
        state is the previous state
        q is a {(state,action):value} dict
        visits is a {(state,action):n} dict. n is how many times action was done in state
        acc_rewards is the accumulated reward

    it observes (s, r) for some world-state s and real reward r
    """
    def __init__(self, env, get_features, discount, explore=0.2, step_size=0.01,
                winit=0, label="SARSA_LFA"):
        """env is the feature environment to interact with
        get_features is a function get_features(state,action) that returns the list of feature values
        discount is the discount factor
        explore is the proportion of time the agent will explore
        step_size is gradient descent step size
        winit is the initial value of the weights
        label is the label for plotting
        """
        RL_agent.__init__(self)
        self.env = env
        self.get_features = get_features
        self.actions = env.actions
        self.discount = discount
        self.explore = explore
        self.step_size = step_size
        self.winit = winit
        self.label = label
        self.restart()
```

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

rlFeatures.py — (continued)

```
    def restart(self):
        """make the agent relearn, and reset the accumulated rewards
        """
        self.acc_rewards = 0
        self.state = self.env.state
        self.features = self.get_features(self.state, list(self.env.actions)[0])
        self.weights = [self.winit for f in self.features]
        self.action = self.select_action(self.state)
```

do takes in the number of steps.

rlFeatures.py — (continued)

```
    def do(self,num_steps=100):
        """do num_steps of interaction with the environment"""
        self.display(2,"s\ta\tr\ts'\tQ\tdelta")
```

```
        for i in range(num_steps):
            next_state,reward = self.env.do(self.action)
            self.acc_rewards += reward
            next_action = self.select_action(next_state)
            feature_values = self.get_features(self.state,self.action)
            oldQ = dot_product(self.weights, feature_values)
            nextQ = dot_product(self.weights, self.get_features(next_state,next_action))
            delta = reward + self.discount * nextQ - oldQ
            for i in range(len(self.weights)):
                self.weights[i] += self.step_size * delta * feature_values[i]
            self.display(2,self.state, self.action, reward, next_state,
                         dot_product(self.weights, feature_values), delta, sep='\t')
            self.state = next_state
            self.action = next_action

    def select_action(self, state):
        """returns an action to carry out for the current agent
        given the state, and the q-function.
        This implements an epsilon-greedy approach
        where self.explore is the probability of exploring.
        """
        if flip(self.explore):
            return random.choice(self.actions)
        else:
            return argmaxe((next_act, dot_product(self.weights,
                                                  self.get_features(state,next_act)))
                                for next_act in self.actions)

    def show_actions(self,state=None):
        """prints the value for each action in a state.
        This may be useful for debugging.
        """
        if state is None:
            state = self.state
        for next_act in self.actions:
            print(next_act,dot_product(self.weights, self.get_features(state,next_act)))

def dot_product(l1,l2):
    return sum(e1*e2 for (e1,e2) in zip(l1,l2))
```

Test code:

rlFeatures.py — (continued)

```
from rlQTest import senv # simple game environment
from rlSimpleGameFeatures import get_features, simp_features
from rlPlot import plot_rl

fa1 = SARSA_LFA_learner(senv, get_features, 0.9, step_size=0.01)
#fa1.max_display_level = 2
#fa1.do(20)
#plot_rl(fa1,steps_explore=10000,steps_exploit=10000,label="SARSA_LFA(0.01)")
```

```
fas1 = SARSA_LFA_learner(senv, simp_features, 0.9, step_size=0.01)
#plot_rl(fas1,steps_explore=10000,steps_exploit=10000,label="SARSA_LFA(simp)")
```

Exercise 12.6 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 12.7 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 12.8 For each of the following first predict, then plot, then explain the behavour 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.

12.5 Multiagent Learning

The next code of for multiple agnets that learn when interacting with other agents. This code is designed to be extended, and as such is restricted to being two agents, a single state, and the only observation is the reward. Coordinating agents can't easily implement that 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, instead we have a controller that tells the egents the percepts (here the percepts are just the reward).

masLearn.py — Simulations of agents learning

```
from display import Displayable
import utilities # argmaxall for (element,value) pairs
import matplotlib.pyplot as plt
import random

class GameAgent(Displayable):
```

```
    next_id=0
    def __init__(self, actions):
        """
        Actions is the set of actions the agent can do. It needs to be told that!
        """
        self.actions = actions
        self.id = GameAgent.next_id
        GameAgent.next_id += 1
        self.display(2,f"Agent {self.id} has actions {actions}")
        self.dist = {act:1 for act in actions} # unnormalized distibution
        self.total_score = 0

    def init_action(self):
        """ The initial action.
        Act randomly initially
        Could be overridden (but I'm not sure why you would).
        """
        self.act = random.choice(self.actions)
        return self.act

    def select_action(self, reward):
        """
        Select the action given the reward.
        This implements "Act randomly" and should be overridden!
        """
        self.total_score += reward
        self.act = random.choice(self.actions)
        return self.act
```

masLearn.py — (continued)

```
class SimpleCountingAgent(GameAgent):
    """This agent just counts the number of times (it thinks) it has won and does the
    actions it thinks is most likely to win.
    """
    def __init__(self, actions, prior_count=1):
        """
        Actions is the set of actions the agent can do. It needs to be told that!
        """
        GameAgent.__init__(self, actions)
        self.prior_count = prior_count
        self.dist = {a: prior_count for a in self.actions} # unnormalized distibution
        self.averew = 0
        self.num_steps = 0

    def select_action(self, reward):
        self.total_score += reward
        self.num_steps += 1
        self.display(2,f"The reward for agent {self.id} was {reward}")
        self.averew = self.averew+(reward-self.averew)/self.num_steps
        if reward>self.averew:
```

```
            self.dist[self.act] += 1
        else:
            for otheract in self.actions:
                if otheract != self.act:
                    self.dist[otheract] += 1/(len(self.actions))
        self.display(2,f"Distribution for agent {self.id} is {normalize(self.dist)}")
        self.act = select_from_dist(self.dist)
        self.display(2,f"Agent {self.id} did {self.act}")
        return self.act
```

masLearn.py — (continued)

```
class SimpleQAgent(GameAgent):
    """This agent maintains the Q-function for each state.
    (Or just the average reward as the future state is all the same).
    Chooses the best action using
    """
    def __init__(self, actions, q_init=100, alpha=0.1, prob_step_size=0.001, min_prob=0.01):
        """
        Actions is the set of actions the agent can do. It needs to be told that!
        q_init is the initial q-values
        alpha is the step size for action estimate
        prob_step_size is the step size for probability change
        min_prob is the minimum a probability should become
        """
        GameAgent.__init__(self, actions)
        self.Q = {a:q_init for a in self.actions}
        self.dist = normalize({a:0.7+random.random() for a in self.actions}) # start with random dist
zero
        self.alpha = alpha
        self.prob_step_size = prob_step_size
        self.min_prob = min_prob
        self.num_steps = 1 # (1 because it isonly used after initial step)

    def select_action(self, reward):
        self.total_score += reward
        self.display(2,f"The reward for agent {self.id} was {reward}")
        self.Q[self.act] += self.alpha*(reward-self.Q[self.act])
        a_best = utilities.argmaxall(self.Q.items())
        for a in self.actions:
            if a in a_best:
                self.dist[a] += self.prob_step_size
            else:
                self.dist[a] -= min(self.dist[a], self.prob_step_size)
                self.dist[a] = max(self.dist[a],self.min_prob)
        self.dist = normalize(self.dist)
        self.display(2,f"Distribution for agent {self.id} is {self.dist}")
        self.act = select_from_dist(self.dist)
        self.display(2,f"Agent {self.id} did {self.act}")
        return self.act

```

```python
def normalize(dist):
    """unnorm dict is a {value:number} dictionary, where the numbers are all non-negative
    returns dict where the numbers sum to one
    """
    tot = sum(dist.values())
    return {var:val/tot for (var,val) in dist.items()}

def select_from_dist(dist):
    rand = random.random()
    for (act,prob) in normalize(dist).items():
        rand -= prob
        if rand < 0:
            return act
```

The simulator takes a game and simulates the game:

masLearn.py — (continued)

```python
class SimulateGame(Displayable):
    def __init__(self, game, agents):
        self.game = game
        self.agents = agents # list of agents
        self.action_history = []
        self.reward_history = []
        self.dist_history = []
        self.actions = tuple(ag.init_action() for ag in self.agents)
        self.num_steps = 0

    def go(self, steps):
        for i in range(steps):
            self.num_steps += 1
            self.rewards = self.game.play(self.actions)
            self.reward_history.append(self.rewards)
            self.actions = tuple(self.agents[i].select_action(self.rewards[i])
                                 for i in range(self.game.num_agents))
            self.action_history.append(self.actions)
            self.dist_history.append([normalize(ag.dist) for ag in self.agents])
        print("Scores:", ' '.join(f"Agent {ag.id} average reward={ag.total_score/self.num_steps}" f
        #return self.reward_history, self.action_history

    def action_dist(self,which_actions=[1,1]):
        """ which actions is [a0,a1]
        returns the empirical disctribition of actions for agents,
           where ai specifies the index of the actions for agent i
        """
        return [sum(1 for a in sim.action_history
                      if a[i]==gm.actions[i][which_actions[i]])/len(sim.action_history)
                  for i in range(2)]
```

masLearn.py — (continued)

```python

    def plot_dynamics(self, x_action=0, y_action=0):
```

```
        plt.ion() # make it interactive
        agents = self.agents
        x_act = self.game.actions[0][x_action]
        y_act = self.game.actions[1][y_action]
        plt.xlabel(f"Action {self.agents[0].actions[x_action]} for Agent {agents[0].id}")
        plt.ylabel(f"Action {self.agents[1].actions[y_action]} for Agent {agents[1].id}")
        plt.plot([self.dist_history[t][0][x_act] for t in range(len(self.dist_history))],
                 [self.dist_history[t][1][y_act] for t in range(len(self.dist_history))])
        #plt.legend()
```

The following are some games from Poole and Mackworth [2017].

masLearn.py — (continued)

```
class ShoppingGame(Displayable):
    def __init__(self):
        self.num_agents = 2
        self.actions = [['shopping', 'football']]*2

    def play(self, actions):
        return {('football', 'football'): (2,1),
                ('football', 'shopping'): (0,0),
                ('shopping', 'football'): (0,0),
                ('shopping', 'shopping'): (1,2)}[actions]


class SoccerGame(Displayable):
    def __init__(self):
        self.num_agents = 2
        self.actions = [['left', 'right']]*2

    def play(self, actions):
        return {('left', 'left'): (0.6, 0.4),
                ('left', 'right'): (0.2, 0.8),
                ('right', 'left'): (0.3, 0.7),
                ('right', 'right'): (0.9,0.1)
               }[actions]

class GameShow(Displayable):
    def __init__(self):
        self.num_agents = 2
        self.actions = [['take', 'give']]*2

    def play(self, actions):
        return {('take', 'take'): (100, 100),
                ('take', 'give'): (1100, 0),
                ('give', 'take'): (0, 1100),
                ('give', 'give'): (1000,1000)
               }[actions]


class UniqueNEGameExample(Displayable):
```

```
    def __init__(self):
        self.num_agents = 2
        self.actions = [['a1', 'b1', 'c1'],['d2', 'e2', 'f2']]

    def play(self, actions):
        return {('a1', 'd2'): (3, 5),
                ('a1', 'e2'): (5, 1),
                ('a1', 'f2'): (1, 2),
                ('b1', 'd2'): (1, 1),
                ('b1', 'e2'): (2, 9),
                ('b1', 'f2'): (6, 4),
                ('c1', 'd2'): (2, 6),
                ('c1', 'e2'): (4, 7),
                ('c1', 'f2'): (0, 8)
               }[actions]

# Choose one:
# gm = ShoppingGame()
# gm = SoccerGame()
# gm = GameShow()
# gm = UniqueNEGameExample()

# Choose one:
# sim=SimulateGame(gm,[SimpleQAgent(gm.actions[0]), SimpleQAgent(gm.actions[1])]); sim.go(10000)
# sim= SimulateGame(gm,[SimpleCountingAgent(gm.actions[0]), SimpleCountingAgent(gm.actions[1])]);
# sim=SimulateGame(gm,[SimpleCountingAgent(gm.actions[0]), SimpleQAgent(gm.actions[1])]); sim.go(1


# sim.plot_dynamics()

# empirical proportion that agents did their action at index 1:
# sim.action_dist([1,1])

# learned distribution for agent 0
# sim.agents[0].dist
```

Chapter 13

Relational Learning

13.1 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 Property-based Collaborative Filtering

```
import random
import matplotlib.pyplot as plt
import urllib.request
from learnProblem import Learner
from display import Displayable

class CF_learner(Learner):
    def __init__(self,
                 rating_set,          # a Rating_set object
                 rating_subset = None, # subset of ratings to be used as training ratings
                 test_subset = None, # subset of ratings to be used as test ratings
                 step_size = 0.01,    # gradient descent step size
                 reglz = 1.0,         # the weight for the regularization terms
                 num_properties = 10,  # number of hidden properties
                 property_range = 0.02 # properties are initialized to be between
                                       # -property_range and property_range
                 ):
        self.rating_set = rating_set
        self.ratings = rating_subset or rating_set.training_ratings # whichever is not empty
        if test_subset is None:
```

```
            self.test_ratings = self.rating_set.test_ratings
        else:
            self.test_ratings = test_subset
        self.step_size = step_size
        self.reglz = reglz
        self.num_properties = num_properties
        self.num_ratings = len(self.ratings)
        self.ave_rating = (sum(r for (u,i,r,t) in self.ratings)
                             /self.num_ratings)
        self.users = {u for (u,i,r,t) in self.ratings}
        self.items = {i for (u,i,r,t) in self.ratings}
        self.user_bias = {u:0 for u in self.users}
        self.item_bias = {i:0 for i in self.items}
        self.user_prop = {u:[random.uniform(-property_range,property_range)
                              for p in range(num_properties)]
                             for u in self.users}
        self.item_prop = {i:[random.uniform(-property_range,property_range)
                              for p in range(num_properties)]
                             for i in self.items}
        self.zeros = [0 for p in range(num_properties)]
        self.iter=0

    def stats(self):
        self.display(1,"ave sumsq error of mean for training=",
                  sum((self.ave_rating-rating)**2 for (user,item,rating,timestamp)
                      in self.ratings)/len(self.ratings))
        self.display(1,"ave sumsq error of mean for test=",
                  sum((self.ave_rating-rating)**2 for (user,item,rating,timestamp)
                      in self.test_ratings)/len(self.test_ratings))
        self.display(1,"error on training set",
                     self.evaluate(self.ratings))
        self.display(1,"error on test set",
                     self.evaluate(self.test_ratings))
```

learn carries out *num_iter* steps of gradient descent.

relnCollFilt.py — (continued)

```
    def prediction(self,user,item):
        """Returns prediction for this user on this item.
        The use of .get() is to handle users or items not in the training set.
        """
        return (self.ave_rating
                +  self.user_bias.get(user,0) #self.user_bias[user]
                + self.item_bias.get(item,0) #self.item_bias[item]
                + sum([self.user_prop.get(user,self.zeros)[p]*self.item_prop.get(item,self.zeros)[p]
                       for p in range(self.num_properties)]))

    def learn(self, num_iter = 50):
        """ do num_iter iterations of gradient descent."""
        for i in range(num_iter):
            self.iter += 1
```

```
            abs_error=0
            sumsq_error=0
            for (user,item,rating,timestamp) in random.sample(self.ratings,len(self.ratings)):
                error = self.prediction(user,item) - rating
                abs_error += abs(error)
                sumsq_error += error * error
                self.user_bias[user] -= self.step_size*error
                self.item_bias[item] -= self.step_size*error
                for p in range(self.num_properties):
                    self.user_prop[user][p] -= self.step_size*error*self.item_prop[item][p]
                    self.item_prop[item][p] -= self.step_size*error*self.user_prop[user][p]
            for user in self.users:
                 self.user_bias[user] -= self.step_size*self.reglz* self.user_bias[user]
                 for p in range(self.num_properties):
                     self.user_prop[user][p] -= self.step_size*self.reglz*self.user_prop[user][p]
            for item in self.items:
                self.item_bias[item] -= self.step_size*self.reglz*self.item_bias[item]
                for p in range(self.num_properties):
                    self.item_prop[item][p] -= self.step_size*self.reglz*self.item_prop[item][p]
            self.display(1,"Iteration",self.iter,
                  "(Ave Abs,AveSumSq) training =",self.evaluate(self.ratings),
                  "test =",self.evaluate(self.test_ratings))
```

evaluate evaluates current predictions on the rating set:

relnCollFilt.py — (continued)

```
    def evaluate(self,ratings):
        """returns (avergage_absolute_error, average_sum_squares_error) for ratings
        """
        abs_error = 0
        sumsq_error = 0
        if not ratings: return (0,0)
        for (user,item,rating,timestamp) in ratings:
            error = self.prediction(user,item) - rating
            abs_error += abs(error)
            sumsq_error += error * error
        return abs_error/len(ratings), sumsq_error/len(ratings)
```

13.1.1 Alternative Formulation

An alternative formulation is to regularize after each update.

13.1.2 Plotting

relnCollFilt.py — (continued)

```
    def plot_predictions(self, examples="test"):
        """
        examples is either "test" or "training" or the actual examples
        """
```

```
        if examples == "test":
            examples = self.test_ratings
        elif examples == "training":
            examples = self.ratings
        plt.ion()
        plt.xlabel("prediction")
        plt.ylabel("cumulative proportion")
        self.actuals = [[] for r in range(0,6)]
        for (user,item,rating,timestamp) in examples:
            self.actuals[rating].append(self.prediction(user,item))
        for rating in range(1,6):
            self.actuals[rating].sort()
            numrat=len(self.actuals[rating])
            yvals = [i/numrat for i in range(numrat)]
            plt.plot(self.actuals[rating], yvals, label="rating="+str(rating))
        plt.legend()
        plt.draw()
```

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

relnCollFilt.py — (continued)

```
    def plot_property(self,
                     p,             # property
                     plot_all=False, # true if all points should be plotted
                     num_points=200 # number of random points plotted if not all
                     ):
        """plot some of the user-movie ratings,
        if plot_all is true
        num_points is the number of points selected at random plotted.

        the plot has the users on the x-axis sorted by their value on property p and
        with the items on the y-axis sorted by their value on property p and
        the ratings plotted at the corresponding x-y position.
        """
        plt.ion()
        plt.xlabel("users")
        plt.ylabel("items")
        user_vals = [self.user_prop[u][p]
                     for u in self.users]
        item_vals = [self.item_prop[i][p]
                     for i in self.items]
        plt.axis([min(user_vals)-0.02,
                  max(user_vals)+0.05,
                  min(item_vals)-0.02,
                  max(item_vals)+0.05])
        if plot_all:
            for (u,i,r,t) in self.ratings:
                plt.text(self.user_prop[u][p],
```

```
                     self.item_prop[i][p],
                     str(r))
        else:
            for i in range(num_points):
                (u,i,r,t) = random.choice(self.ratings)
                plt.text(self.user_prop[u][p],
                         self.item_prop[i][p],
                         str(r))
        plt.show()
```

13.1.3 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)

```
class Rating_set(Displayable):
    def __init__(self,
                 date_split=892000000,
                 local_file=False,
                 url="http://files.grouplens.org/datasets/movielens/ml-100k/u.data",
                 file_name="u.data"):
        self.display(1,"reading...")
        if local_file:
            lines = open(file_name,'r')
        else:
            lines = (line.decode('utf-8') for line in urllib.request.urlopen(url))
        all_ratings = (tuple(int(e) for e in line.strip().split('\t'))
                       for line in lines)
        self.training_ratings = []
        self.training_stats = {1:0, 2:0, 3:0, 4:0 ,5:0}
        self.test_ratings = []
        self.test_stats = {1:0, 2:0, 3:0, 4:0 ,5:0}
        for rate in all_ratings:
            if rate[3] < date_split: # rate[3] is timestamp
                self.training_ratings.append(rate)
                self.training_stats[rate[2]] += 1
            else:
                self.test_ratings.append(rate)
                self.test_stats[rate[2]] += 1
        self.display(1,"...read:", len(self.training_ratings),"training ratings and",
                len(self.test_ratings),"test ratings")
        tr_users = {user for (user,item,rating,timestamp) in self.training_ratings}
        test_users = {user for (user,item,rating,timestamp) in self.test_ratings}
        self.display(1,"users:",len(tr_users),"training,",len(test_users),"test,",
                     len(tr_users & test_users),"in common")
        tr_items = {item for (user,item,rating,timestamp) in self.training_ratings}
        test_items = {item for (user,item,rating,timestamp) in self.test_ratings}
        self.display(1,"items:",len(tr_items),"training,",len(test_items),"test,",
```

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

Sometimes it is useful to plot a property for all (*user*, *item*, *rating*) triples. There are too many such triples in the data set. The method *create_top_subset* creates a much smaller dataset where this makes sense. It picks the most rated items, then picks the users who have the most ratings on these items. It is designed for depicting the meaning of properties, and may not be useful for other purposes.

relnCollFilt.py — (continued)

```
    def create_top_subset(self, num_items = 30, num_users = 30):
        """Returns a subset of the ratings by picking the most rated items,
        and then the users that have most ratings on these, and then all of the
        ratings that involve these users and items.
        """
        items = {item for (user,item,rating,timestamp) in self.training_ratings}

        item_counts = {i:0 for i in items}
        for (user,item,rating,timestamp) in self.training_ratings:
            item_counts[item] += 1

        items_sorted = sorted((item_counts[i],i) for i in items)
        top_items = items_sorted[-num_items:]
        set_top_items = set(item for (count, item) in top_items)

        users = {user for (user,item,rating,timestamp) in self.training_ratings}
        user_counts = {u:0 for u in users}
        for (user,item,rating,timestamp) in self.training_ratings:
            if item in set_top_items:
                user_counts[user] += 1

        users_sorted = sorted((user_counts[u],u)
                              for u in users)
        top_users = users_sorted[-num_users:]
        set_top_users = set(user for (count, user) in top_users)
        used_ratings = [ (user,item,rating,timestamp)
                        for (user,item,rating,timestamp) in self.training_ratings
                        if user in set_top_users and item in set_top_items]
        return used_ratings

movielens = Rating_set()
learner1 = CF_learner(movielens, num_properties = 1)
#learner1.learn(50)
# learner1.plot_predictions(examples = "training")
# learner1.plot_predictions(examples = "test")
#learner1.plot_property(0)
#movielens_subset = movielens.create_top_subset(num_items = 20, num_users = 20)
#learner_s = CF_learner(movielens, rating_subset=movielens_subset, test_subset=[], num_properties=1)
#learner_s.learn(1000)
```

```
#learner_s.plot_property(0,plot_all=True)
```

Chapter 14

Version History

- 2021-05-13 Version 0.9.0 Major revisions to chapters 8 and 9. Introduced recursive conditioning, simplified much code. New section on multiagent reinforcement learning.
- 2020-11-04 Version 0.8.6 simplified value iteration for MDPs.
- 2020-10-20 Version 0.8.4 planning simplified, and gives error if goal not part of state (by design). Fixed arc costs.
- 2020-07-21 Version 0.8.2 added positions and string to constraints
- 2019-09-17 Version 0.8.0 rerepresented blocks world (Section 6.1.2) due to bug found by Donato Meoli.

Bibliography

Lichman, M. (2013), UCI machine learning repository. URL http://archive.ics.uci.edu/ml. 107

Poole, D. L. and Mackworth, A. K. (2017), *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2nd edition, URL https://artint.info. 148

Index