Python code for Artificial Intelligence: Foundations of Computational Agents

David L. Poole and Alan K. Mackworth

Version 0.9.7 of September 15, 2023.

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

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

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

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

1.1 Why Python?

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

Python is reasonably efficient. Efficiency is usually not a problem for small examples. If your Python code is not efficient enough, a general procedure to improve it is to find out what is taking most of the time, and implement just that part more efficiently in some lower-level language. 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 larger projects.

1.2 Getting Python

You need Python 3.9 or later[1] (`https://python.org/`) and a compatible version of matplotlib (`https://matplotlib.org/`). This code is *not* compatible with Python 2 (e.g., with Python 2.7).

Download and install the latest Python 3 release from `https://python.org/` or `https://www.anaconda.com/download`. 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** (`https://ipython.org/`) [Pérez and Granger, 2007]. 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 python3 (or perhaps just ipython or python) from a shell.

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

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

[1]The only feature of 3.9 used is dictionary union. The feature of 3.8 used is `:=`. To use earlier versions 3.8, replace `|` with `dict_union` defined in Section 1.7.4.

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

You can then interact at the last prompt.

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

The rest of this chapter is about what is special about the code for AI tools. We 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/might happen given certain conditions. In many such cases, we don't want side effects. When an agent acts in the world, side effects are appropriate.

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

1.5 Features of Python

1.5.1 f-strings

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

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

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

where `e1` and `e2` are expressions, is an abbreviation for

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

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

1.5.2 Lists, Tuples, Sets, Dictionaries and Comprehensions

We make extensive uses of lists, tuples, sets and dictionaries (dicts). See `https://docs.python.org/3/library/stdtypes.html`

One of the nice features of Python is the use of **comprehensions**[2] (and also list, tuple, set and dictionary comprehensions). A generator expression is of the form

(*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* is a variable (or a pattern that can be on the left side of =), *iter* is an iterator, which can generate a stream of data, such as a list, a set, a range object (to enumerate integers between ranges) or a file. *cond* is an expression that evaluates to either True or False for each *e*, and *fe* is an expression that will be evaluated for each value of *e* for which *cond* returns *True*.

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)
```

[2] `https://docs.python.org/3/reference/expressions.html#displays-for-lists-sets-and-dictionaries`

```
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* is a built-in function that, given a dictionary, returns an iterator of $(index, value)$ pairs.

1.5.3 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 *i*th function in the list is meant to add *i* to its argument:[3]

[3]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.

pythonDemo.py — Some tricky examples

```python
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)

```python
# 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 *fun*2 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 *fun*1 uses late binding, and *fun*2 uses early binding. *fun_list*3 and *fun_list*4 are equivalent to the first two (except *fun_list*4 uses a different *i* variable).

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.

1.5.4 Generators

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

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

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

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, f"only positive steps implemented in myrange: {step}"
    i = start
    while i<stop:
        yield i
        i += step

print("list(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.) There is not need to make it with indexing.

Yield can be used to generate the same sequence of values as in the example of Section 1.5.2:

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 1.5.2.

It is straightforward to write a version of the built-in *enumerate* called *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 **run time** of the program. The most straightforward way to compute run time 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 interactive python shell, 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 run time is at least 0.2 seconds.

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

1.6.2 Plotting: Matplotlib

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

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

pythonDemo.py — (continued)

```
import matplotlib.pyplot as plt

def myplot(minv,maxv,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(minv,maxv,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

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

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 or instances

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

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)

```python
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

```python
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 arg 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. $flip(p)$ returns True with probability p, and otherwise returns False.

utilities.py — (continued)

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

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

utilities.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(f"{item_prob_dist} is not a probability
        distribution")
```

1.7.4 Dictionary Union

This is now | in Python 3.9, has been be replaced in the code. Use this if you want to back-port to an older version of Python.

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

utilities.py — (continued)

```
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 done here. In particular, you should also test 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

Agent Architectures and Hierarchical Control

This implements the controllers described in Chapter 2 of Poole and Mackworth [2023].

These provide sequential implementations of the control. More sophisticated version may have them run concurrently (either as coroutines or in parallel).

In this version the higher-levels call the lower-levels. 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

In the initial implementation, both agents and the envoronment are treated as objects in the send of object-oriented programs: they can have an internal state they maintain, and can evaluate methods that can provide answers. This is the same representation used for the reinforcement learning algorithms (Chapter 13).

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

An **agent** takes the precept, updates its internal state, and output it next action. An agent implements the method `select_action` that takes percept and returns its next action.

The methods `do` and `select_action` are chained together to build a simulator. In order to start this, we need either an action or a percept. There are two variants used:

- An agent implements the `initial_action()` method which is used initially. This is the method used in the reinforcement learning chapter (page 275).
- The environment implements the `initial_percept()` method which gives the initial percept. This is the method used in this chapter.

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 percept and the actions are represented as variable-value dictionaries. When agent has only a limited number of actions, the action can be a single value.

In the following code `raise NotImplementedError()` is a way to specify an abstract method that needs to be overridden in any implemented agent or environment.

agents.py — Agent and Controllers

```python
from display import Displayable

class Agent(Displayable):

    def initial_action(self, percept):
        """return the initial action"""
        raise NotImplementedError("go") # abstract method

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

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

Note that *Environment* is a subclass of *Displayable* so that it can use the *display* method described in Section 1.7.1.

agents.py — (continued)

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

    def do(self, action):
        """does the action in the environment
```

```
        returns the next percept """
        raise NotImplementedError("Environment.do") # abstract method
```

The simulator lets the agent and the environment take turns in updating their states and returning the action and the percept.

The first implementation is a simple procedure to carry out n steps of the simulation and return the agent state and the environment state at the end.

agents.py — (continued)

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

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

2.2 Paper buying agent and environment

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

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

2.2.1 The Environment

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

Here we assume that the prices are obtained from the *prices* list (which cycles) 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.

agentBuying.py — Paper-buying agent

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

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_percept(self):
        """return initial percept"""
        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 percept consiting of price and
            instock"""
        used = pick_from_dist({6:0.1, 5:0.1, 4:0.1, 3:0.3, 2:0.2, 1:0.2})
        # used = pick_from_dist({7:0.1, 6:0.2, 5:0.2, 4:0.3, 3:0.1, 2:0.1})
            # uses more paper
        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}
```

2.2.2 The Agent

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

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

agentBuying.py — (continued)

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

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

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

agentBuying.py — (continued)

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

2.2.3 Plotting

The following plots the price and number in stock history:

agentBuying.py — (continued)

```
import matplotlib.pyplot as plt

class Plot_history(object):
```

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


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

    def plot_agent_hist(self):
        """plot history of buying"""
        num = len(ag.buy_history)
        plt.bar(range(1,num+1), ag.buy_history, label="Bought")
        plt.legend()
        #plt.draw()

# pl = Plot_history(ag,env)
# sim.go(90)
#pl.plot_env_hist()
#pl.plot_agent_hist()
```

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

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

- Justify a performance measure that is a fair comparison. Note that minimizing the total amount of money spent may be unfair to agents who have built up a stockpile, and favors agents that end up with no paper.
- Give a controller that can work for many different price histories. An agent can use other local state variables, but does not have access to the environment model.
- Is it worthwhile trying to infer the amount of paper that the home uses? (Try your controller with the different paper consumption commented out in `TP_env.do`.)

2.3 Hierarchical Controller

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

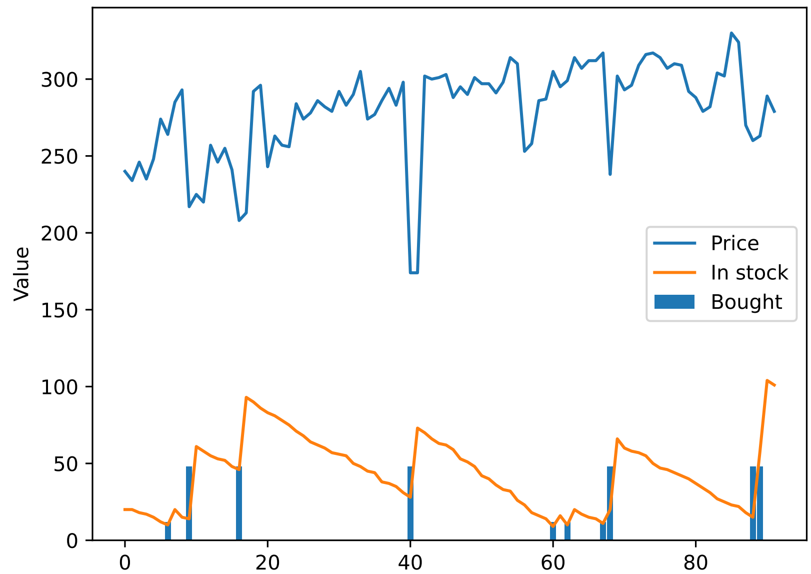


Figure 2.1: Percept and command traces for the paper-buying agent

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 display import Displayable
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 percept(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_percept = percept # use percept function for initial percept too

    def do(self,action):
        """ action is {'steer':direction}
        direction is 'left', 'right' or 'straight'
        """
        if self.crashed:
            return self.percept()
        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.percept()
```

The Boolean `whisker` method returns True when the whisker and the wall intersect.

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.percept = env.initial_percept()
        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_percept(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.percept = self.env.do({"steer":self.steer(target_pos)})
            remaining -= 1
            arrived = self.close_enough(target_pos)
        return {'arrived':arrived}
```

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

agentMiddle.py — (continued)

```
    def steer(self,target_pos):
        if self.percept['whisker']:
            self.display(3,'whisker on', self.percept)
            return "left"
        else:
            return self.head_towards(target_pos)

    def head_towards(self,target_pos):
            """ given a target position, return the action that heads
                towards that position
            """
            gx,gy = target_pos
            rx,ry = self.percept['rob_x_pos'],self.percept['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.percept['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.percept['rob_x_pos'],self.percept['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 display import Displayable
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(Displayable):
    def __init__(self, body,top):
        """sets up the plot
        """
        self.body = body
        self.top = top
        plt.ion()
        plt.axes().set_aspect('equal')
        self.redraw()

    def redraw(self):
        plt.clf()
        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]
```

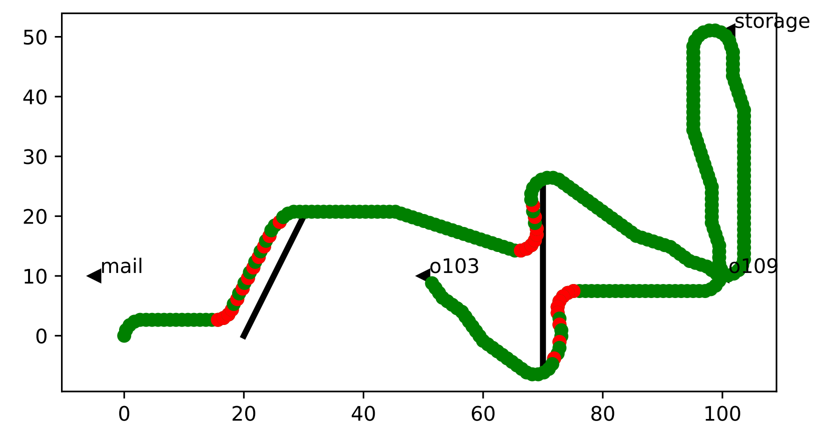


Figure 2.2: A trace of the trajectory of the agent. Red dots correspond to the whisker sensor being on; the green dot to the whisker sensor being off. The agent starts at position $(0,0)$ facing up.

```
                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.gca().figure.canvas.draw()
            if self.body.history or self.body.wall_history:
                self.plot_run()

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

The following code plots the agent as it acts in the world. Figure 2.2 shows the result of the top.do

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:
```

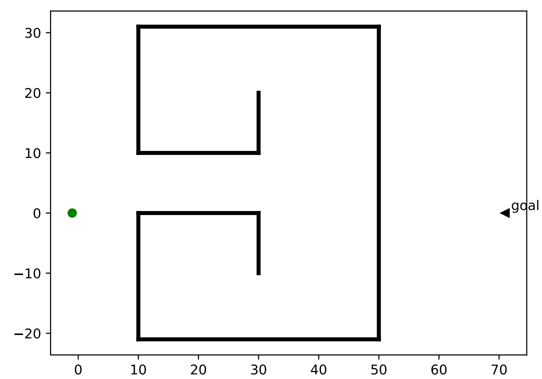


Figure 2.3: Robot trap

```
# 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 2.2 The following code implements a robot trap (Figure 2.3). Write a controller that can escape the "trap" and get to the goal. See Exercise 2.4 in the 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']})
```

Plotting for Moving Targets

Exercise 2.5 refers to targets that can move. The following implements targets than can be moved by the user (using the mouse).

agentFollowTarget.py — Plotting for moving targets

```
import matplotlib.pyplot as plt
from agentTop import Plot_env, body, top
```

```

class Plot_follow(Plot_env):
    def __init__(self, body, top, epsilon=2.5):
        """plot the agent in the environment.
        epsilon is the threshold how how close someone needs to click to
            select a location.
        """
        Plot_env.__init__(self, body, top)
        self.epsilon = epsilon
        self.canvas = plt.gca().figure.canvas
        self.canvas.mpl_connect('button_press_event', self.on_press)
        self.canvas.mpl_connect('button_release_event', self.on_release)
        self.canvas.mpl_connect('motion_notify_event', self.on_move)
        self.pressloc = None
        self.pressevent = None
        for loc in self.top.locations:
            self.display(2,f"  loc {loc} at {self.top.locations[loc]}")

    def on_press(self, event):
        self.display(2,'v',end="")
        self.display(2,f"Press at ({event.xdata},{event.ydata}")
        for loc in self.top.locations:
            lx,ly = self.top.locations[loc]
            if abs(event.xdata- lx) <= self.epsilon and abs(event.ydata-
                ly) <= self.epsilon :
                self.pressloc = loc
                self.pressevent = event
                self.display(2,"moving",loc)

    def on_release(self, event):
        self.display(2,'^',end="")
        if self.pressloc is not None: #and event.inaxes ==
            self.pressevent.inaxes:
            self.top.locations[self.pressloc] = (event.xdata, event.ydata)
            self.display(1,f"Placing {self.pressloc} at {(event.xdata,
                event.ydata)}")
        self.pressloc = None
        self.pressevent = None

    def on_move(self, event):
        if self.pressloc is not None: # and event.inaxes ==
            self.pressevent.inaxes:
            self.display(2,'-',end="")
            self.top.locations[self.pressloc] = (event.xdata, event.ydata)
            self.redraw()
        else:
            self.display(2,'.',end="")

# try:
# pl=Plot_follow(body,top)
```

```
# top.do({'visit':['o109','storage','o109','o103']})
```

Exercise 2.3 Change the code to also allow walls to move.

Chapter 3

Searching for Solutions

3.1 Representing Search Problems

A search problem consists of:

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

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

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

searchProblem.py — representations of search problems

```
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 (or enumeration) of the arcs for the neighbors of node"""
        raise NotImplementedError("neighbors") # abstract method

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

The neighbors is a list of arcs. A (directed) arc consists of a *from_node* node and a *to_node* node. The arc is the pair ⟨*from_node*, *to_node*⟩, but can also contain a non-negative *cost* (which defaults to 1) and can be labeled with an *action*.

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):
        self.from_node = from_node
        self.to_node = to_node
        self.action = action
        self.cost=cost
        assert cost >= 0, (f"Cost cannot be negative: {self}, cost={cost}")

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

3.1.1 Explicit Representation of Search Graph

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

An **explicit graph** consists of

- a list or set of nodes

- a list or set of arcs
- a start node
- a list or set of goal nodes
- (optionally) a 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 (a list of arcs)"""
        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 += f"{arc}. "
        return res
```

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. Nore that we only use the most basic form of Python's `yield` for enumerations (Section 1.5.4).

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 enumerates the nodes in the path from the last elements backwards.
        """
        current = self
        while current.arc is not None:
            yield current.arc.to_node
            current = current.initial
        yield current.initial

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

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

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','B',3), Arc('A','C',1), Arc('B','D',1), Arc('B','G',3),
         Arc('C','B',1), Arc('C','D',3), Arc('D','G',1)],
    start = 'A',
    goals = {'G'},
    positions={'A': (0, 2), 'B': (1, 1), 'C': (0,1), 'D': (1,0), 'G': (2,0)})
```

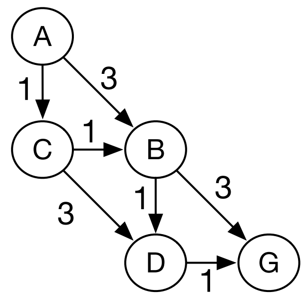


Figure 3.1: problem1

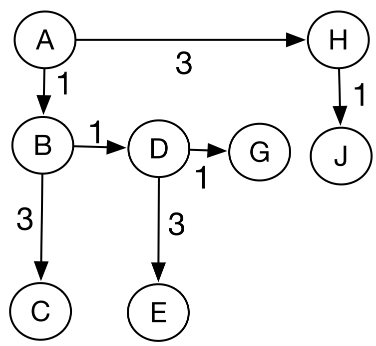


Figure 3.2: problem2

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)

```
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(
```

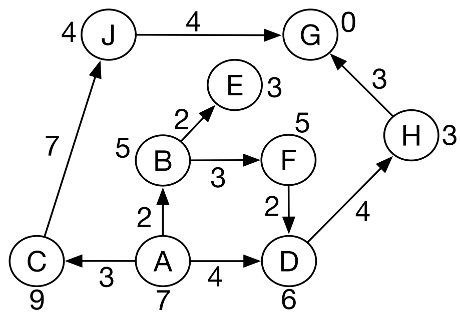


Figure 3.3: `simp_delivery_graph` with arc costs and h values of nodes

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

The `simp_delivery_graph` is the graph shown Figure 3.3. This is Figure 3.3 with the heuristics of Figure 3.1 as shown in Fugure 3.13 of [Poole and Mackworth, 2023],

searchProblem.py — (continued)

```
simp_delivery_graph = Search_problem_from_explicit_graph(
    {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J'},
    [   Arc('A', 'B', 2),
        Arc('A', 'C', 3),
        Arc('A', 'D', 4),
        Arc('B', 'E', 2),
        Arc('B', 'F', 3),
        Arc('C', 'J', 7),
        Arc('D', 'H', 4),
        Arc('F', 'D', 2),
        Arc('H', 'G', 3),
        Arc('J', 'G', 4)],
   start = 'A',
   goals = {'G'},
   hmap = {
        'A': 7,
        'B': 5,
        'C': 9,
        'D': 6,
        'E': 3,
        'F': 5,
        'G': 0,
        'H': 3,
        'J': 4,
    })
```

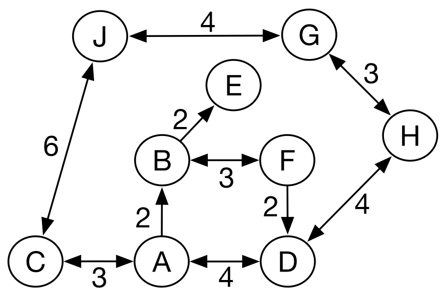


Figure 3.4: cyclic_simp_delivery_graph with arc costs

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

searchProblem.py — (continued)

```
cyclic_simp_delivery_graph = Search_problem_from_explicit_graph(
    {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J'},
    [   Arc('A', 'B', 2),
        Arc('A', 'C', 3),
        Arc('A', 'D', 4),
        Arc('B', 'A', 2),
        Arc('B', 'E', 2),
        Arc('B', 'F', 3),
        Arc('C', 'A', 3),
        Arc('C', 'J', 7),
        Arc('D', 'A', 4),
        Arc('D', 'H', 4),
        Arc('F', 'B', 3),
        Arc('F', 'D', 2),
        Arc('G', 'H', 3),
        Arc('G', 'J', 4),
        Arc('H', 'D', 4),
        Arc('H', 'G', 3),
        Arc('J', 'C', 6),
        Arc('J', 'G', 4)],
   start = 'A',
   goals = {'G'},
   hmap = {
        'A': 7,
        'B': 5,
        'C': 9,
        'D': 6,
        'E': 3,
        'F': 5,
        'G': 0,
```

```
        'H': 3,
        'J': 4,
    })
```

3.2 Generic Searcher and Variants

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

3.2.1 Searcher

A *Searcher* for a problem can be asked repeatedly for the next path. To solve a problem, you 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 neighbors so that it implements depth-first search in an intuitive manner (expanding the first neighbor first). The call to *list* is for the case when the neighbors are generated (and not already in a list). Reversing the neighbors might not be required for other methods. The calls to *reversed* and *list* can be removed, and the algorithm still implements depth-fist search.

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

searchGeneric.py — (continued)

```
# Depth-first search for problem1; do the following:
# searcher1 = Searcher(searchProblem.problem1)
# searcher1.search() # find first solution
# searcher1.search() # find next solution (repeat until no solutions)
# searcher_sdg = Searcher(searchProblem.simp_delivery_graph)
# searcher_sdg.search() # find first or next solution
```

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

3.2.2 Frontier as a Priority Queue

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

Following the lead of the Python documentation, `http://docs.python.org/3.9/library/heapq.html`, a frontier is a list of triples. The first element of each triple is the value to be minimized. The second element is a unique index which specifies the order that the elemnets were added to the queue, and the third element is the path that is on the queue. The use of the unique index ensures that the priority queue implementation does not compare paths; whether one path is less than another is not defined. It also lets us control what sort of search (e.g., depth-first or breadth-first) occurs when the value to be minimized does not give a unique next path.

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

searchGeneric.py — (continued)

```
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 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 problem 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)   # what needs to be changed to make this succeed?
    test(AStarSearcher)

# example queries:
# searcher1 = Searcher(searchProblem.simp_delivery_graph) # DFS
# searcher1.search() # find first path
# searcher1.search() # find next path
# searcher2 = AStarSearcher(searchProblem.simp_delivery_graph) # A*
# searcher2.search() # find first path
# searcher2.search() # find next path
# searcher3 = Searcher(searchProblem.cyclic_simp_delivery_graph) # DFS
# searcher3.search() # find first path with DFS. What do you expect to
    happen?
# searcher4 = AStarSearcher(searchProblem.cyclic_simp_delivery_graph) # 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_simp_delivery_graph)
# 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 cycle if `path.end() in path.initial_nodes()`) Compare no pruning, multiple path pruning and cycle pruning for the cyclic delivery problem. Which works better in terms of number of paths expanded, computational time or space?

3.3 Branch-and-bound Search

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

Depth-first search methods do not need 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 neighbors can be generated.

Here is a unit test and some queries:

searchBranchAndBound.py — (continued)

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

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

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

Exercise 3.7 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 3.8 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.simp_delivery_graph,"Acyclic Delivery")
#   run(searchProblem.cyclic_simp_delivery_graph,"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 Variables

A **variable** consists of a name, a domain and an optional (x,y) position (for displaying). The domain of a variable is a list or a tuple, as the ordering will matter in the representation of constraints.

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

```
import random
import matplotlib.pyplot as plt

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})"
```

4.1.2 Constraints

A **constraint** consists of:

- A tuple (or list) of variables is called the **scope**.
- A **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.
- An optional name
- An optional (x, y) position

cspProblem.py — Representations of a Constraint Satisfaction Problem

```
from variable import Variable

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

    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 can_evaluate(self, assignment):
        """
        assignment is a variable:value dictionary
        returns True if the constraint can be evaluated given assignment
        """
        return all(v in assignment for v in self.scope)

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

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

4.1.3 CSPs

A constraint satisfaction problem (CSP) requires:

- *variables*: a list or set of variables
- *constraints*: a set or list of constraints.

Other properties are inferred from these:

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

cspProblem.py — (continued)

```
class CSP(object):
    """A CSP consists of
    * a title (a string)
    * variables, a set of variables
    * constraints, a list of constraints
    * var_to_const, a variable to set of constraints dictionary
    """
    def __init__(self, title, variables, constraints):
        """title is a string
        variables is set of variables
        constraints is a list of constraints
```

```
        """
        self.title = title
        self.variables = variables
        self.constraints = constraints
        self.var_to_const = {var:set() for var in self.variables}
        for con in constraints:
            for var in con.scope:
                self.var_to_const[var].add(con)

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

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

csp.*consistent*(*assignment*) returns true if the assignment is consistent with each of the constraints in *csp* (i.e., all of the constraints that can be evaluated evaluate to true). 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 con.can_evaluate(assignment))
```

The **show** method uses matplotlib to show the graphical structure of a constraint network. If the node positions are not specified, this gives different positions each time it is run; if you don't like the graph, try again.

cspProblem.py — (continued)

```
    def show(self):
        plt.ion()  # interactive
        ax = plt.figure().gca()
        ax.set_axis_off()
        plt.title(self.title)
        var_bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5")
        con_bbox = dict(boxstyle="square,pad=1.0",color="green")
        for var in self.variables:
            if var.position is None:
                var.position = (random.random(), random.random())
        for con in self.constraints:
            if con.position is None:
```

```
                con.position = tuple(sum(var.position[i] for var in con.scope)/len(con.scope)
                                          for i in range(2))
            bbox = dict(boxstyle="square,pad=1.0",color="green")
            for var in con.scope:
                ax.annotate(con.string, var.position, xytext=con.position,
                                    arrowprops={'arrowstyle':'-'},bbox=con_bbox,
                                    ha='center')
        for var in self.variables:
            x,y = var.position
            plt.text(x,y,var.name,bbox=var_bbox,ha='center')
```

4.1.4 Examples

In the following code *ne_*, when given a number, returns a function that is true when its argument is not that number. For example, if $f = ne\_(3)$, then $f(2)$ is True and $f(3)$ is False. That is, $ne\_(x)(y)$ is true when $x \neq y$. Allowing a function of multiple arguments to use its arguments one at a time is called **currying**, after the logician Haskell Curry. Functions used as conditions in constraints require names (so they can be printed).

cspExamples.py — Example CSPs

```
from cspProblem import Variable, CSP, Constraint
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__ = f"{val} != "   # name of the function
    return nev
```

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

cspExamples.py — (continued)

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

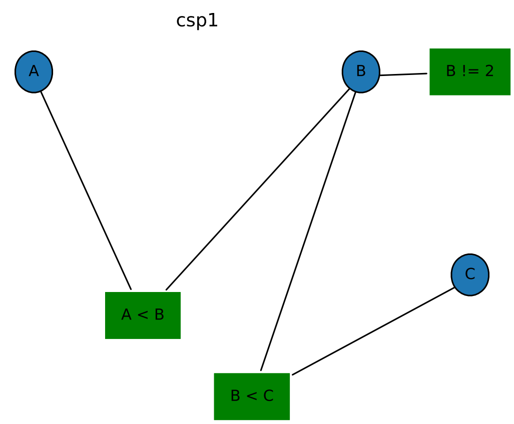


Figure 4.1: csp1.show()

```
X = Variable('X', {1,2,3})
Y = Variable('Y', {1,2,3})
Z = Variable('Z', {1,2,3})
csp0 = CSP("csp0", {X,Y,Z},
          [ 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. The CSP *csp*1*s* is the same, but with only the constraints $A < B$ and $B < C$

cspExamples.py — (continued)

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

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

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

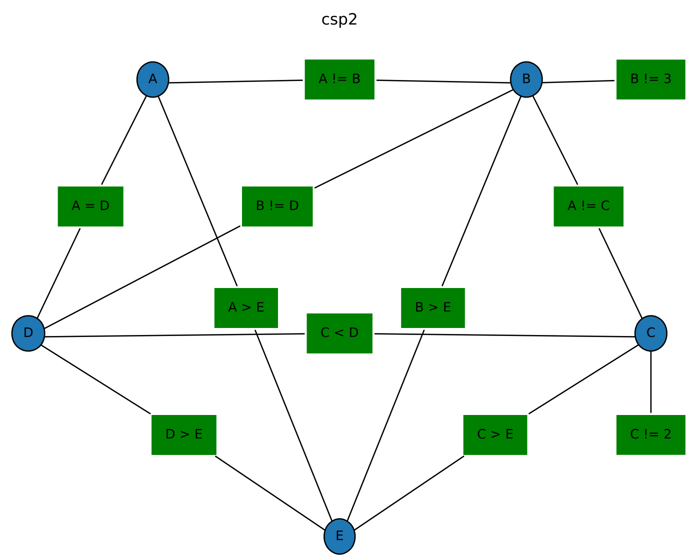


Figure 4.2: csp2.show()

two more.

cspExamples.py — (continued)

```
D = Variable('D', {1,2,3,4}, position=(0,0.4))
E = Variable('E', {1,2,3,4}, position=(0.5,0))
csp2 = CSP("csp2", {A,B,C,D,E},
           [ Constraint([B], ne_(3), "B != 3", position=(1,0.9)),
            Constraint([C], ne_(2), "C != 2", position=(1,0.2)),
            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([E,A], lt, "E < A"),
            Constraint([E,B], lt, "E < B"),
            Constraint([E,C], lt, "E < C"),
            Constraint([E,D], lt, "E < D"),
            Constraint([B,D], ne, "B != D")])
```

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

cspExamples.py — (continued)

```
csp3 = CSP("csp3", {A,B,C,D,E},
           [Constraint([A,B], ne, "A != B"),
            Constraint([A,D], lt, "A < D"),
```

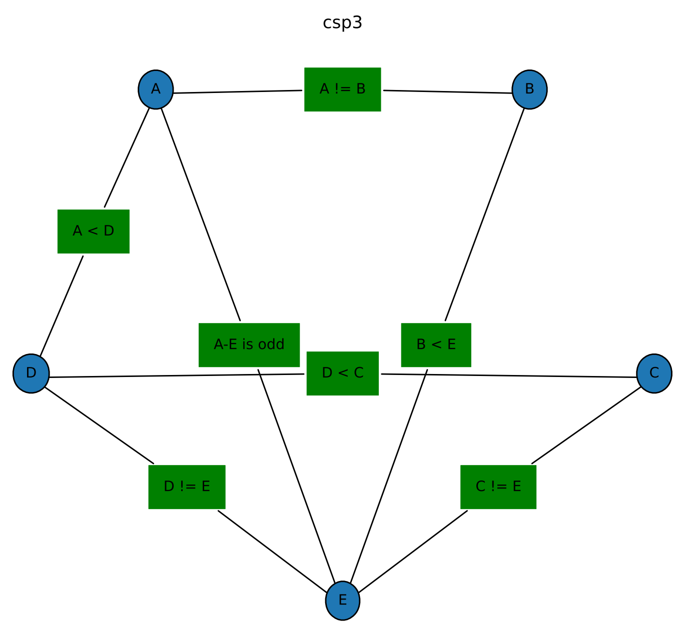


Figure 4.3: csp3.show()

```
            Constraint([A,E], lambda a,e: (a-e)%2 == 1, "A-E is odd"),
            Constraint([B,E], lt, "B < E"),
            Constraint([D,C], lt, "D < C"),
            Constraint([C,E], ne, "C != E"),
            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("csp4", {A,B,C,D,E},
           [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, "B != D"),
            Constraint([C,E], ne, "C != E")])
```

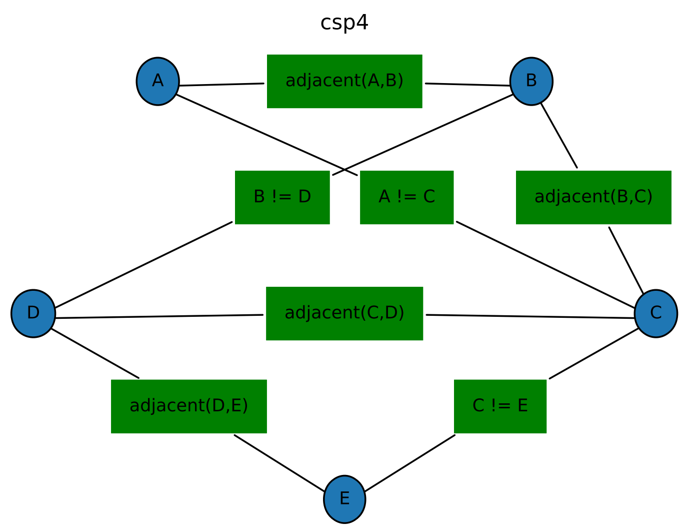
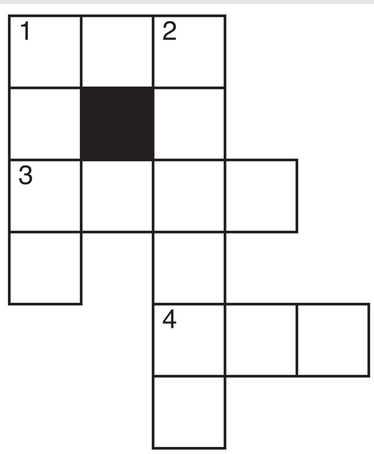


Figure 4.4: csp4.show()



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

Figure 4.5: crossword1: a crossword puzzle to be solved

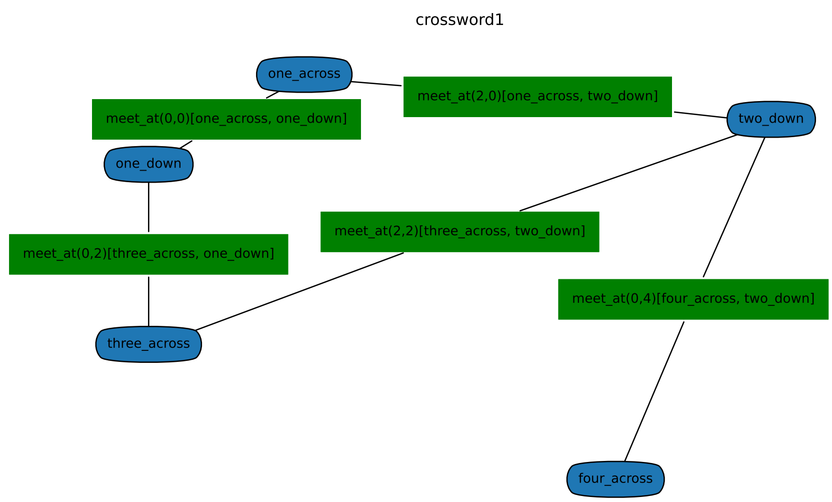


Figure 4.6: crossword1.show()

The following examples represent the crossword shown in Figure 4.5.

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

cspExamples.py — (continued)

```
def meet_at(p1,p2):
    """returns a function of two words that is true
                when the words intersect at positions 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__ = f"meet_at({p1},{p2})"
    return meets

one_across = Variable('one_across', {'ant', 'big', 'bus', 'car', 'has'},
    position=(0.3,0.9))
one_down = Variable('one_down', {'book', 'buys', 'hold', 'lane', 'year'},
    position=(0.1,0.7))
two_down = Variable('two_down', {'ginger', 'search', 'symbol', 'syntax'},
```

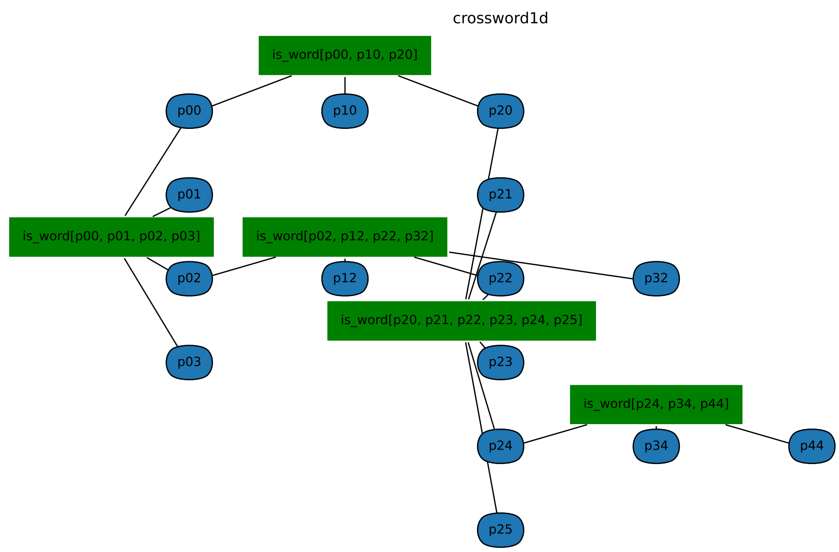


Figure 4.7: crossword1d.show()

```
    position=(0.9,0.8))
three_across = Variable('three_across', {'book', 'buys', 'hold', 'land',
    'year'}, position=(0.1,0.3))
four_across = Variable('four_across',{'ant', 'big', 'bus', 'car', 'has'},
    position=(0.7,0.0))
crossword1 = CSP("crossword1",
                {one_across, one_down, two_down, three_across,
                    four_across},
                [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. This is shown in Figure 4.7.

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"}

# pij is the variable representing the letter i from the left and j down
    (starting from 0)
p00 = Variable('p00', letters, position=(0.1,0.85))
p10 = Variable('p10', letters, position=(0.3,0.85))
p20 = Variable('p20', letters, position=(0.5,0.85))
p01 = Variable('p01', letters, position=(0.1,0.7))
p21 = Variable('p21', letters, position=(0.5,0.7))
p02 = Variable('p02', letters, position=(0.1,0.55))
p12 = Variable('p12', letters, position=(0.3,0.55))
p22 = Variable('p22', letters, position=(0.5,0.55))
p32 = Variable('p32', letters, position=(0.7,0.55))
p03 = Variable('p03', letters, position=(0.1,0.4))
p23 = Variable('p23', letters, position=(0.5,0.4))
p24 = Variable('p24', letters, position=(0.5,0.25))
p34 = Variable('p34', letters, position=(0.7,0.25))
p44 = Variable('p44', letters, position=(0.9,0.25))
p25 = Variable('p25', letters, position=(0.5,0.1))

crossword1d = CSP("crossword1d",
                {p00, p10, p20, # first row
                 p01, p21, # second row
                 p02, p12, p22, p32, # third row
                 p03, p23, #fourth row
                 p24, p34, p44, # fifth row
                 p25 # sixth row
                 },
                [Constraint([p00, p10, p20], is_word,
                    position=(0.3,0.95)), #1-across
                 Constraint([p00, p01, p02, p03], is_word,
                     position=(0,0.625)), # 1-down
                 Constraint([p02, p12, p22, p32], is_word,
                     position=(0.3,0.625)), # 3-across
                 Constraint([p20, p21, p22, p23, p24, p25], is_word,
                     position=(0.45,0.475)), # 2-down
                 Constraint([p24, p34, p44], is_word,
                     position=(0.7,0.325)) # 4-across
                 ])
```

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

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

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

cspExamples.py — (continued)

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

def n_queens(n):
    """returns a CSP for n-queens"""
    columns = list(range(n))
    variables = [Variable(f"R{i}",columns) for i in range(n)]
    return CSP("n-queens",
               variables,
                [Constraint([variables[i], variables[j]], queens(i,j))
                     for i in range(n) for j in range(n) if i != j])

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

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

Unit tests

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

cspExamples.py — (continued)

```
def test_csp(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, f"Solution not correct for {csp}"
    print("Passed unit test")
```

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

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

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

4.2 A Simple Depth-first Solver

The first solver carries out a depth-first search through the space of partial assignments. This takes in a CSP problem and an optional variable ordering (a list of the variables in the CSP). It returns a generator of the solutions (see Section 1.5.4 on `yield` for enumerations).

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

```
from cspExamples import csp1,csp1s,csp2,test_csp, crossword1, crossword1d

def dfs_solver(constraints, context, var_order):
    """generator for all solutions to csp.
    context is an assignment of values to some of the variables.
    var_order is  a list of the variables in csp that are not in context.
    """
    to_eval = {c for c in constraints if c.can_evaluate(context)}
    if all(c.holds(context) for c in to_eval):
        if var_order == []:
            yield context
        else:
            rem_cons = [c for c in constraints if c not in to_eval]
            var = var_order[0]
            for val in var.domain:
                yield from dfs_solver(rem_cons, context|{var:val},
                    var_order[1:])

def dfs_solve_all(csp, var_order=None):
    """depth-first CSP solver to return a list of all solutions to csp.
    """
    if var_order == None:  # use an arbitrary variable order
        var_order = list(csp.variables)
    return list( dfs_solver(csp.constraints, {}, var_order))

def dfs_solve1(csp, var_order=None):
    """depth-first CSP solver to find single solution or None if there are
        no solutions.
    """
    if var_order == None:  # use an arbitrary variable order
        var_order = list(csp.variables)
```

```
    gen = dfs_solver(csp.constraints, {}, var_order)
    try:    # Python generators raise an exception if there are no more
        elements.
        return next(gen)
    except StopIteration:
        return None

if __name__ == "__main__":
    test_csp(dfs_solve1)

#Try:
# dfs_solve_all(csp1)
# dfs_solve_all(csp2)
# dfs_solve_all(crossword1)
# dfs_solve_all(crossword1d) # warning: may take a *very* long time!
```

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

Exercise 4.7 Estimate how long `dfs_solve_all(crossword1d)` will take on your computer. To do this, reduce the number of variables that need to be assigned, so that the simplifies problem can be solved in a reasonable time (between 0.1 second and 10 seconds). This can be done by reducing the number of variables in `var_order`, as the program only splits on these. How much more time will it take if the number of variables is increased by 1? (Try it!) Then extrapolate to all of the variables. See Section 1.6.1 for how to time your code. Would making the code 100 times faster or using a computer 100 times faster help?

4.3 Converting CSPs to Search Problems

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

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

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

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

```python
from cspProblem import CSP, Constraint
from searchProblem import Arc, Search_problem

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 neighbors.

cspSearch.py — (continued)

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

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

cspSearch.py — (continued)

```python
from cspExamples import csp1,csp1a,csp2,test_csp, crossword1, crossword1d
from searchGeneric import Searcher
```

```

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

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

## Test Solving CSPs with Search:
searcher1 = Searcher(Search_from_CSP(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 4.8 What would happen if we constructed the new assignment by assigning $node[var] = val$ (with side effects) instead of using dictionary union? Give an example of where this could give a wrong answer. How could the algorithm be changed to work with side effects? (Hint: think about what information needs to be in a node).

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

4.4 Consistency Algorithms

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

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

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

```
from display import Displayable

class Con_solver(Displayable):
    """Solves a CSP with arc consistency and domain splitting
    """
    def __init__(self, csp, **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 = {var:var.domain for var in self.csp.variables}
        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. For pedagogical purposes, a user interface could allow the user to select an arc. Alternatively a more sophisticated selection could be employed.

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 consistent with the assignment to the other variables. To make it easier to understand, the following code treats unary (with no other variables in the constraint) and binary (with one other variables in the constraint) constraints as special cases. These cases are not strictly necessary; the last case covers the first two cases, but is more difficult to understand without seeing the first two cases.

```
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*.

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
        """
        if ind == len(other_vars):
            return const.holds(env)
        else:
            var = other_vars[ind]
            for val in domains[var]:
                if self.any_holds(domains, const, env|{var:val}, other_vars, ind + 1):
                    return True
            return False
```

4.4.1 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
        """
        new_domains = self.make_arc_consistent(domains, to_do)
        if any(len(new_domains[var]) == 0 for var in new_domains):
            return False
        elif all(len(new_domains[var]) == 1 for var in new_domains):
            self.display(2, "solution:", {var: select(
                new_domains[var]) for var in new_domains})
            return {var: select(new_domains[var]) for var in new_domains}
        else:
            var = self.select_var(x for x in self.csp.variables if len(new_domains[x]) > 1)
            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)
```

```python

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)

```python
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)

```python
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 4.10 Implement of *solve_all* that is like *solve_one* but returns the set of all solutions.

Exercise 4.11 Implement *solve_enum* that enumerates the solutions. It should use Python's *yield* (and perhaps *yield from*).

Unit test:

cspConsistency.py — (continued)

```python
from cspExamples import test_csp
def ac_solver(csp):
    "arc consistency (solve_one)"
```

```
    return Con_solver(csp).solve_one()

if __name__ == "__main__":
    test_csp(ac_solver)
```

4.4.2 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 4.12 When splitting a domain, this code splits the domain into half, approximately in half (without any effort to make a sensible choice). Does it work better to split one element from a domain?

Unit test:

cspConsistency.py — (continued)

```
from cspExamples import test_csp
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_csp(ac_search_solver)
```

Testing:

cspConsistency.py — (continued)

```
from cspExamples import csp1, csp1s, 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())
#searcher4c = Searcher(Search_with_AC_from_CSP(crossword1d))
#print(searcher4c.search())
```

4.5 Solving CSPs using Stochastic Local Search

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

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

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

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

cspSLS.py — Stochastic Local Search for Solving CSPs

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

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

    A node is a variable:value dictionary"""
    def __init__(self, csp):
        self.csp = csp
        self.variables_to_select = {var for var in self.csp.variables
                                    if len(var.domain) > 1}
        # 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_choice(var.domain) for
                                   var in self.csp.variables}
        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, which is not appropriate for CSPs with many variables, as it is a relatively expensive operation for these cases.

This method selects one of two implementations. The argument *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 variable (one in most conflict) is selected
        prob_anycon is the probability that a variable 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 4.13 This does an initial random assignment but does not do any random restarts. Implement a searcher that takes in the maximum number of walk steps (corresponding to existing *max_steps*) and the maximum number of restarts, and returns the total number of steps for the first solution found. (As in *search*, the solution found can be extracted from the variable *self.current_assignment*).

4.5.1 Any-conflict

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_choice(self.conflicts) # pick random conflict
                var = random_choice(con.scope) # pick variable in conflict
            else:
                var = random_choice(self.variables_to_select)
            if len(var.domain) > 1:
                val = random_choice([val for val in var.domain
                                  if val is not
                                      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 4.14 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.

4.5.2 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. When a variable `var` is assigned a value `val`, for each constraint that has become satisfied or unsatisfied, each variable involved in the constraint need to have it's count updates. The change is recorded in the dictionary *var_differential*, which is used to update the priority queue (see Section 4.5.3).

cspSLS.py — (continued)

```python
    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_choice(self.conflicts)
                var = random_choice(con.scope)
            else: #pick any variable that can be selected
                var = random_choice(self.variables_to_select)
            if len(var.domain) > 1: # var has other values
                ## Pick a value
                val = random_choice([val for val in var.domain if val is not
                                    self.current_assignment[var]])
                self.display(2,"Assigning",var,val)
                ## Update the priority queue
                var_differential = {}
                self.current_assignment[var]=val
                for varcon in self.csp.var_to_const[var]:
                    self.display(3,"Checking",varcon)
                    if varcon.holds(self.current_assignment):
                        if varcon in self.conflicts: #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_choice(st):
    """selects a random element from set st.
```

```
    It would be more efficient to convert to a tuple or list only once
    (left as exercise)."""
    return random.choice(tuple(st))
```

Exercise 4.15 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 4.16 These implementations always select a value for the variable selected that is different from its current value (if that is possible). Change the code so that it does not have this restriction (so it can leave the value the same). Would you expect this code to be faster? Does it work worse (or better)?

4.5.3 Updatable Priority Queues

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

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

cspSLS.py — (continued)

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

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

    def add(self,elt,val):
        """adds elt to the priority queue with priority=val.
        """
        assert val <= 0,val
```

```
        assert elt not in self.elt_map, elt
        new_triple = [val, random.random(),elt]
        heapq.heappush(self.pq, new_triple)
        self.elt_map[elt] = new_triple

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

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

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

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

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

4.5.4 Plotting Run-Time Distributions

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

cspSLS.py — (continued)

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

class Runtime_distribution(object):
    def __init__(self, csp, xscale='log'):
        """Sets up plotting for csp
        xscale is either 'linear' or 'log'
        """
        self.csp = csp
        plt.ion()
        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")
        SLSearcher.max_display_level= temp_mdl #restore display
```

Figure 4.8 gives run-time distributions for 3 algorithms. It is also useful to compare the distributions of different runs of the same algorithms and settings.

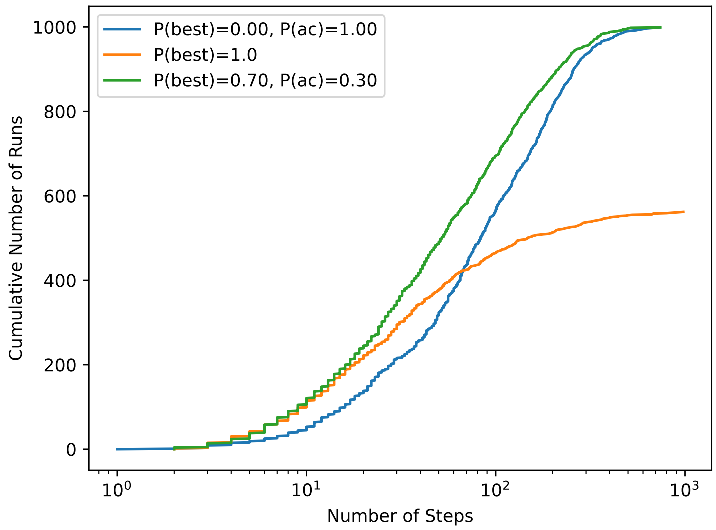


Figure 4.8: Run-time distributions for three algorithms on *csp*2.

4.5.5 Testing

cspSLS.py — (continued)

```
from cspExamples import test_csp
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_csp(sls_solver)
    test_csp(any_conflict_solver)

from cspExamples import csp1, csp1s, csp2, crossword1, crossword1d

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

4.6 Discrete Optimization

A `SoftConstraint` is a constraint, but where the condition is a real-valued function. Because the definition of the constraint class did not force the condition to be Boolean, you can use the `Constraint` class for soft constraints too.

cspSoft.py — Representations of Soft Constraints

```
from cspProblem import Variable, Constraint, CSP
class SoftConstraint(Constraint):
    """A Constraint consists of
    * scope: a tuple of variables
    * function: a real-valued 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, function, string=None, position=None):
        Constraint.__init__(self, scope, function, string, position)

    def value(self,assignment):
        return self.holds(assignment)
```

cspSoft.py — (continued)

```
A = Variable('A', {1,2}, position=(0.2,0.9))
B = Variable('B', {1,2,3}, position=(0.8,0.9))
C = Variable('C', {1,2}, position=(0.5,0.5))
D = Variable('D', {1,2}, position=(0.8,0.1))

def c1fun(a,b):
    if a==1: return (5 if b==1 else 2)
    else: return (0 if b==1 else 4 if b==2 else 3)
```

```
c1 = SoftConstraint([A,B],c1fun,"c1")
def c2fun(b,c):
    if b==1: return (5 if c==1 else 2)
    elif b==2: return (0 if c==1 else 4)
    else: return (2 if c==1 else 0)
c2 = SoftConstraint([B,C],c2fun,"c2")
def c3fun(b,d):
    if b==1: return (3 if d==1 else 0)
    elif b==2: return 2
    else: return (2 if d==1 else 4)
c3 = SoftConstraint([B,D],c3fun,"c3")

def penalty_if_same(pen):
    "returns a function that gives a penalty of pen if the arguments are
        the same"
    return lambda x,y: (pen if (x==y) else 0)

c4 = SoftConstraint([C,A],penalty_if_same(3),"c4")

scsp1 = CSP("scsp1", {A,B,C,D}, [c1,c2,c3,c4])

### The second soft CSP has an extra variable, and 2 constraints
E = Variable('E', {1,2}, position=(0.1,0.1))

c5 = SoftConstraint([C,E],penalty_if_same(3),"c5")
c6 = SoftConstraint([D,E],penalty_if_same(2),"c6")
scsp2 = CSP("scsp1", {A,B,C,D,E}, [c1,c2,c3,c4,c5,c6])
```

4.6.1 Branch-and-bound Search

Here we specialize the branch-and-bound algorithm (Section 3.3 on page 53) to solve soft CSP problems.

cspSoft.py — (continued)

```
from display import Displayable, visualize
import math

class DF_branch_and_bound_opt(Displayable):
    """returns a branch and bound searcher for a problem.
    An optimal assignment with cost less than bound can be found by calling
        search()
    """
    def __init__(self, csp, bound=math.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__()
```

```
        self.csp = csp
        self.best_asst = None
        self.bound = bound

    def optimize(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.num_expanded=0
        self.cbsearch({}, 0, self.csp.constraints)
        self.display(1,"Number of paths expanded:",self.num_expanded)
        return self.best_asst, self.bound

    def cbsearch(self, asst, cost, constraints):
        """finds the optimal solution that extends path and is less the bound"""
        self.display(2,"cbsearch:",asst,cost,constraints)
        can_eval = [c for c in constraints if c.can_evaluate(asst)]
        rem_cons = [c for c in constraints if c not in can_eval]
        newcost = cost + sum(c.value(asst) for c in can_eval)
        self.display(2,"Evaluaing:",can_eval,"cost:",newcost)
        if newcost < self.bound:
            self.num_expanded += 1
            if rem_cons==[]:
                self.best_asst = asst
                self.bound = newcost
                self.display(1,"New best assignment:",asst," cost:",newcost)
            else:
                var = next(var for var in self.csp.variables if var not in asst)
                for val in var.domain:
                    self.cbsearch({var:val}|asst, newcost, rem_cons)

# bnb = DF_branch_and_bound_opt(scsp1)
# bnb.max_display_level=3 # show more detail
# bnb.optimize()
```

Exercise 4.18 Change the stochastic-local search algorithms to work for soft constraints. Hint: The analog of a conflict is a soft constraint that is not at its lowest value. Instead of the number of constraints violated, consider how much a change in a variable affects the objective function. Instead of returning a solution, return the best assignment found.

Chapter 5

Propositions and Inference

5.1 Representing Knowledge Bases

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

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 __repr__(self):
        """returns the string representation of a clause.
        """
        if self.body:
            return self.head + " <- " + " & ".join(str(a) for a in
                 self.body) + "\n"
        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:
            self.add_clause(c)

    def add_clause(self, c):
        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)
```

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

logicProblem.py — (continued)

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

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

# print(kb)
```

5.2 Bottom-up Proofs (with askables)

fixed_point computes the fixed point of the knowledge base *kb*.

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

```
from logicProblem import yes

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

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

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

logicBottomUp.py — (continued)

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

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

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

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

Exercise 5.2 This algorithm runs in time $O(n^2)$, where n is the number of clauses, for a bounded number of elements in the body; each iteration goes through each of the clauses, and in the worst case, it will do an iteration for each clause. It is possible to implement this in time $O(n)$ time by creating an index that maps an atom to the set of clauses with that atom in the body. Implement this. What is its complexity as a function of n and b, the maximum number of atoms in the body of a clause?

Exercise 5.3 It is possible to be asymptotically more efficient (in terms of 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 (with askables)

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

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

```
from logicProblem import yes

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

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

logicTopDown.py — (continued)

```
from logicProblem import triv_KB
def test():
    a1 = prove(triv_KB,['i_am'])
    assert a1, f"triv_KB proving i_am gave {a1}"
    a2 = prove(triv_KB,['i_smell'])
    assert not a2, f"triv_KB proving i_smell gave {a2}"
    print("Passed unit tests")
if __name__ == "__main__":
    test()
# try
from logicProblem import elect
```

```
# elect.max_display_level=3 # give detailed trace
# prove(elect,['live_w6'])
# prove(elect,['lit_l1'])
```

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

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

5.4 Debugging and Explanation

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

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

logicExplain.py — Explaining Proof Procedure for Definite Clauses

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

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

def prove_body(kb, ans_body, indent=""):
    """returns proof tree if kb |- ans_body or "fail" if there is no proof
    ans_body is a list of atoms in a body to be proved
    """
    proofs = []
    for atom in ans_body:
```

```
        proof_at = prove_atom(kb, atom, indent+" ")
        if proof_at == "fail":
            return "fail" # fail if any proof fails
        else:
            proofs.append(proof_at)
    return proofs
```

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

logicExplain.py — (continued)

```
from logicProblem import triv_KB
def test():
    a1 = prove_atom(triv_KB,'i_am')
    assert a1, f"triv_KB proving i_am gave {a1}"
    a2 = prove_atom(triv_KB,'i_smell')
    assert a2=="fail", "triv_KB proving i_smell gave {a2}"
    print("Passed unit tests")
if __name__ == "__main__":
    test()
# try
from logicProblem import elect, elect_bug
# elect.max_display_level=3 # give detailed trace
# prove_atom(elect, 'live_w6')
# prove_atom(elect, 'lit_l1')
```

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

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

logicExplain.py — (continued)

```
helptext = """Commands are:
ask atom     ask is there is a proof for atom (atom should not be in quotes)
how          show the clause that was used to prove atom
how n        show the clause used to prove the nth element of the body
up           go back up proof tree to explore other parts of the proof tree
kb           print the knowledge base
quit         quit this interaction (and go back to Python)
help         print this text
"""

def interact(kb):
    going = True
    ups = []  # stack for going up
```

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

def print_rule(proof):
    (head,body) = proof
    if body == "answered":
        print(head,"was answered yes")
    elif body == []:
             print(head,"is a fact")
```

```
    else:
           print(head,"<-")
           for i,a in enumerate(body):
               print(i,":",a[0])

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

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

```
>>> interact(elect)
logicExplain: ask lit_l1
Is up_s2 true? no
Is down_s2 true? yes
Is down_s1 true? yes
yes
logicExplain: how
lit_l1 <-
0 : light_l1
1 : live_l1
2 : ok_l1
logicExplain: how 1
live_l1 <-
0 : live_w0
logicExplain: how 0
live_w0 <-
0 : down_s2
1 : live_w2
logicExplain: how 0
down_s2 was answered yes
logicExplain: up
live_w0 <-
0 : down_s2
1 : live_w2
logicExplain: how 1
live_w2 <-
0 : down_s1
1 : live_w3
logicExplain: quit
>>>
```

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

5.5 Assumables

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

logicAssumables.py — Definite clauses with assumables

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

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

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

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

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

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

logicAssumables.py — (continued)

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

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

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

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

logicAssumables.py — (continued)

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

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

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

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

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

logicAssumables.py — (continued)

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

Test cases:

logicAssumables.py — (continued)

```
electa = KBA([
```

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

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

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

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

5.6 Negation-as-failure

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

logicNegation.py — Propositional negation-as-failure

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

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

    def atom(self):
        return self.theatom

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

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

logicNegation.py — (continued)

```
def prove_naf(kb, ans_body, indent=""):
    """ prove with negation-as-failure and askables
    returns True if kb |- ans_body
    ans_body is a list of atoms to be proved
    """
    kb.display(2,indent,'yes <-',' & '.join(str(e) for e in ans_body))
    if ans_body:
        selected = ans_body[0] # select first atom from ans_body
        if isinstance(selected, Not):
            kb.display(2,indent,f"proving {selected.atom()}")
            if prove_naf(kb, [selected.atom()], indent):
                kb.display(2,indent,f"{selected.atom()} succeeded so
                    Not({selected.atom()}) fails")
                return False
            else:
                kb.display(2,indent,f"{selected.atom()} fails so
                    Not({selected.atom()}) succeeds")
                return prove_naf(kb, ans_body[1:],indent+" ")
        if selected in kb.askables:
            return (yes(input("Is "+selected+" true? "))
                    and  prove_naf(kb,ans_body[1:],indent+" "))
        else:
            return any(prove_naf(kb,cl.body+ans_body[1:],indent+" ")
                       for cl in kb.clauses_for_atom(selected))
    else:
        return True   # empty body is true
```

Test cases:

logicNegation.py — (continued)

```
triv_KB_naf = KB([
    Clause('i_am', ['i_think']),
    Clause('i_think'),
    Clause('i_smell', ['i_am', Not('dead')]),
    Clause('i_bad', ['i_am', Not('i_think')])
    ])

triv_KB_naf.max_display_level = 4
def test():
    a1 = prove_naf(triv_KB_naf,['i_smell'])
    assert a1, f"triv_KB_naf proving i_smell gave {a1}"
    a2 = prove_naf(triv_KB_naf,['i_bad'])
    assert not a2, f"triv_KB_naf proving i_bad gave {a2}"
    print("Passed unit tests")
if __name__ == "__main__":
    test()
```

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

logicNegation.py — (continued)

```
beach_KB = KB([
   Clause('away_from_beach', [Not('on_beach')]),
   Clause('beach_access', ['on_beach', Not('ab_beach_access')]),
   Clause('swim_at_beach', ['beach_access', Not('ab_swim_at_beach')]),
   Clause('ab_swim_at_beach', ['enclosed_bay', 'big_city',
       Not('ab_no_swimming_near_city')]),
   Clause('ab_no_swimming_near_city', ['in_BC', Not('ab_BC_beaches')])
    ])

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

Chapter 6

Deterministic Planning

6.1 Representing Actions and Planning Problems

The STRIPS representation of an action consists of:

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

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, feature_domain_dict, actions):
        """Problem domain
        feature_domain_dict is a feature:domain dictionary,
                mapping each feature to its domain
        actions
        """
        self.feature_domain_dict = feature_domain_dict
        self.actions = actions
```

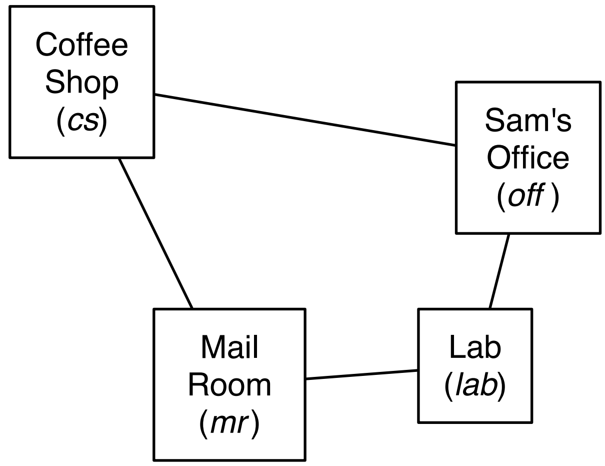
A planning problem consists of a planning domain, an initial state, and a goal. The goal does not need to fully specify the final state.

stripsProblem.py — (continued)

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

6.1.1 Robot Delivery Domain

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



Features to describe states

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

Actions

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

Figure 6.1: Robot Delivery Domain

stripsProblem.py — (continued)

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

stripsProblem.py — (continued)

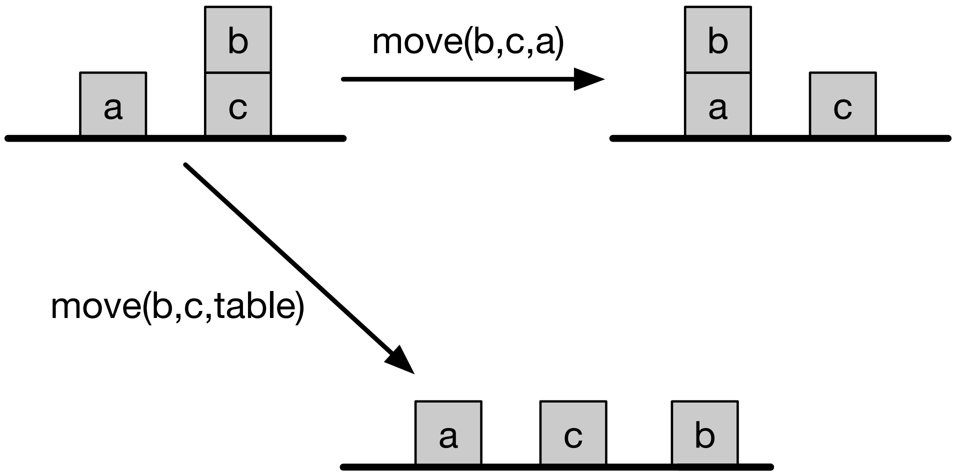


Figure 6.2: Blocks world with two actions

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

6.1.2 Blocks World

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

A state is defined by the two features:

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

There is one parameterized action

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

To handle parameterized actions (which depend on the blocks involved), the actions and the features are all strings, created for 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})
    feature_domain_dict = {on(x):blocks_and_table-{x} for x in blocks}
    feature_domain_dict.update({clear(x):boolean for x in blocks_and_table})
    return STRIPS_domain(feature_domain_dict, stmap)
```

The problem *blocks*1 is a classic example, with 3 blocks, and the goal consists of two conditions. See Figure 6.3. 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'})
```

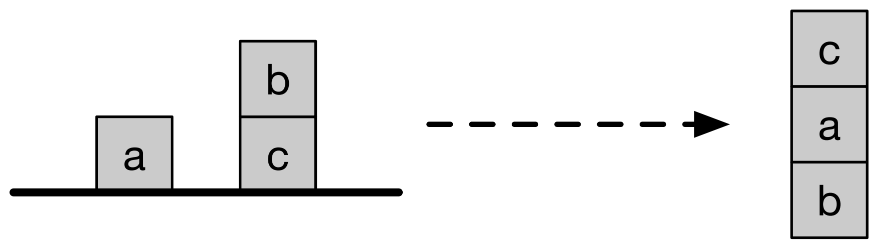


Figure 6.3: Blocks problem blocks1

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

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

stripsProblem.py — (continued)

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

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

Exercise 6.2 Represent the domain so that $on(x, y)$ is a Boolean feature that is True when x is on y, Does the representation of the state need to 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 39), 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 heuristics 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 For more than one start-state/goal combination, test the forward planner with a heuristic function of just $h1$, with just $h2$ and with both. Explain why each one prunes or doesn't prune the search space.

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

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

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

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

Exercise 6.6 Create an admissible heuristic for the blocks world.

6.3 Regression Planning

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

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

A *Subgoal* 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

```
from searchProblem import Arc, Search_problem

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

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

stripsRegressionPlanner.py — (continued)

```
from stripsForwardPlanner import zero

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

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

```python

    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 $\{a : True, b : False\}$ has been visited, then any node that is a superset, e.g., $\{a : True, b : False, d : True\}$, need not be expanded. If the simpler subgoal does not lead to a solution, the more complicated one 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, $\{'MW' : True, 'RHM' : True\}$ is an incompatible assignment. This information may be useful information for a planner; there is no point in trying to achieve these together. Define a subclass of *STRIPS_domain* that can accept a list of incompatible assignments. Modify the regression planner code to use such a list of incompatible assignments. Give an example where the search space is smaller.

Exercise 6.9 After completing the previous exercise, design incompatible assignments for the blocks world. (This 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
```

```python

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 *h*1 and with just *h*2 (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

```python
from cspProblem import Variable, CSP, Constraint

class CSP_from_STRIPS(CSP):
    """A CSP where:
    * CSP variables are constructed for each feature and time, and each
        action and time
    * 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.action_vars[t] is the action variable for time t
        self.action_vars = [Variable(f"Action{t}", prob_domain.actions)
                            for t in range(number_stages)]
        # feat_time_var[f][t] is the variable for feature f at time t
        feat_time_var = {feat: [Variable(f"{feat}_{t}",dom)
                                    for t in range(number_stages+1)]
```

```
                        for (feat,dom) in
                            prob_domain.feature_domain_dict.items()}

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

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

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

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

        constraints += [Constraint((feat_time_var[feat][t],
            self.action_vars[t], feat_time_var[feat][t+1]),
                              eq_if_not_in_({act for act in
                                  prob_domain.actions
                                            if feat in act.effects}))
                          for feat in prob_domain.feature_domain_dict
                          for t in range(number_stages) ]
        variables = set(self.action_vars) | {feat_time_var[feat][t]
                                      for feat in
                                          prob_domain.feature_domain_dict
                                      for t in range(number_stages+1)}
        CSP.__init__(self, variables, constraints)

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

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

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

Note that the underscore ('_') is part of the name; 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__ = f"value_is_{val}"
    return is_fun

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

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

Putting it together, this returns a list of actions that solves the problem *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()) # returns path to solution

## 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()) # returns path to solution

## 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_runs(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 f"{self.action}#{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 features, e.g. *height* $> 1.9m$ might be a Boolean feature constructed from the real-values feature *height*. The next chapter is about neural networdks and how to learn features; in this chapter we construct explicitly 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 provides the base cases for many algorithms (e.g., decision tree algorithm) and baselines that more sophisticated algorithms need to beat. It also provides ways to test various predictors.
- Decision tree learning: one of the classic and simplest learning algorithms, which is the basis of many other algorithms.
- Cross validation and parameter tuning: methods to prevent overfitting.
- Linear regression and classification: other classic and simple techniques that often work well (particularly combined with feature learning or engineering).
- Boosting: combining simpler learning methods to make even better learners.

A good source of classic datasets is the UCI Machine Learning Repository [Lichman, 2013] [Dua and Graff, 2017]. The SPECT, IRIS, and car datasets (carbook is a Boolean version of the car dataset) are from this repository.

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

Figure 7.1: Some of the datasets used here.

7.1 Representations of Data and Predictions

The code uses the following definitions and conventions:

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

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

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

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

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

learnProblem.py — A Learning Problem

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

boolean = [False, True]
```

When creating a dataset, 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. A dataset `ds` has the following attributes

`ds.train` a list of the training examples

`ds.test` a list of the test examples

`ds.target_index` the index of the target

`ds.target` the feature corresponding to the target (a function as described above)

`ds.input_features` a list of the input features

learnProblem.py — (continued)

```
class Data_set(Displayable):
    """ A dataset consists of a list of training data and a list of test
        data.
    """

    def __init__(self, train, test=None, prob_test=0.20, target_index=0,
                   header=None, target_type= None, seed=None): #12345):
        """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
        target_type is either None for automatic detection of target type
             or one of "numeric", "boolean", "cartegorical"
        seed is for random number; None gives a different test set each time
        """
        if seed: # given seed makes partition consistent from run-to-run
            random.seed(seed)
        if test is None:
            train,test = partition_data(train, prob_test)
        self.train = train
        self.test = test

        self.display(1,"Training set has",len(train),"examples. Number of
            columns: ",{len(e) for e in train})
        self.display(1,"Test set has",len(test),"examples. Number of
            columns: ",{len(e) for e in test})
        self.prob_test = prob_test
        self.num_properties = len(self.train[0])
        if target_index < 0: #allows for -1, -2, etc.
            self.target_index = self.num_properties + target_index
        else:
            self.target_index = target_index
        self.header = header
        self.domains = [set() for i in range(self.num_properties)]
        for example in self.train:
            for ind,val in enumerate(example):
```

```
                self.domains[ind].add(val)
        self.conditions_cache = {} # cache for computed conditions
        self.create_features()
        if target_type:
            self.target.ftype = target_type
        self.display(1,"There are",len(self.input_features),"input
            features")

    def __str__(self):
        if self.train and len(self.train)>0:
            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.")
```

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

learnProblem.py — (continued)

```
    def create_features(self):
        """create the set of features
        """
        self.target = None
        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 = list(self.domains[i])
            feat.ftype = self.infer_type(feat.frange)
            if i == self.target_index:
                self.target = feat
            else:
                self.input_features.append(feat)
```

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

learnProblem.py — (continued)

```
    def infer_type(self,domain):
        """Infers the type of a feature with domain
        """
        if all(v in {True,False} for v in domain):
            return "boolean"
```

```
        if all(isinstance(v,(float,int)) for v in domain):
            return "numeric"
        else:
            return "categorical"
```

7.1.1 Creating Boolean Conditions from Features

Some of the algorithms require Boolean input features or features with range $\{0, 1\}$. In order to be able to use these algorithms on datasets that allow for arbitrary domains of input variables, we construct Boolean conditions from the attributes.

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 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.

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

learnProblem.py — (continued)

```
    def conditions(self, max_num_cuts=8, categorical_only = False):
        """returns a set of boolean conditions from the input features
        max_num_cuts is the maximum number of cute for numeric features
        categorical_only is true if only categorical features are made
            binary
        """
        if (max_num_cuts, categorical_only) in self.conditions_cache:
            return self.conditions_cache[(max_num_cuts, categorical_only)]
        conds = []
        for ind,frange in enumerate(self.domains):
            if ind != self.target_index and len(frange)>1:
                if len(frange) == 2:
                    # two values, the feature is equality to one of them.
                    true_val = list(frange)[1] # choose one as true
```

```
                    def feat(e, i=ind, tv=true_val):
                        return e[i]==tv
                    if self.header:
                        feat.__doc__ = f"{self.header[ind]}=={true_val}"
                    else:
                        feat.__doc__ = f"e[{ind}]=={true_val}"
                    feat.frange = boolean
                    feat.ftype = "boolean"
                    conds.append(feat)
                elif all(isinstance(val,(int,float)) for val in frange):
                    if categorical_only: # numeric, don't make cuts
                        def feat(e, i=ind):
                            return e[i]
                        feat.__doc__ = f"e[{ind}]"
                        conds.append(feat)
                    else:
                        # all numeric, create cuts of the data
                        sorted_frange = sorted(frange)
                        num_cuts = min(max_num_cuts,len(frange))
                        cut_positions = [len(frange)*i//num_cuts for i in range(1,num_cuts)]
                        for cut in cut_positions:
                            cutat = sorted_frange[cut]
                            def feat(e, ind_=ind, cutat=cutat):
                                return e[ind_] < cutat

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

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

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

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

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

7.1.2 Evaluating Predictions

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

A **loss** takes a prediction and the actual value and returns a non-negative real number; lower is better. The **error** for a dataset is either the mean loss, or sometimes the sum of the losses. When reporting results the mean is usually used. When it is the sum, this will be made explicit.

The function *evaluate_dataset* returns the average error for each example, where the error for each example depends on the evaluation criteria. 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 log loss (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 features.
        error_measure(prediction,actual) -> non-negative real
        """
        if data:
            try:
                value = statistics.mean(error_measure(predictor(e),
                    self.target(e))
                          for e in data)
            except ValueError: # if error_measure gives an error
                return float("inf") # infinity
            return value
        else:
            return math.nan # not a number
```

The following evaluation criteria are defined. This is defined using a class, `Evaluate` but no instances will be created. Just use `Evaluate.squared_loss` etc.

(Please keep the __doc__ strings a consistent length as they are used in tables.) The prediction is either a real value or a $\{value : probability\}$ dictionary or a list. The actual is either a real number or a key of the prediction.

learnProblem.py — (continued)

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

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

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

    def log_loss(prediction, actual):
        "log loss (bits)"
        try:
            if isinstance(prediction, (list,dict)):
                 return -math.log2(prediction[actual])
            else:
                return -math.log2(prediction) if actual==1 else
                    -math.log2(1-prediction)
        except ValueError:
            return float("inf") # infinity

    def accuracy(prediction, actual):
        "accuracy    "
        if isinstance(prediction, dict):
            prev_val = prediction[actual]
            return 1 if all(prev_val >= v for v in prediction.values())
                else 0
        if isinstance(prediction, list):
            prev_val = prediction[actual]
            return 1 if all(prev_val >= v for v in prediction) else 0
        else:
            return 1 if abs(actual-prediction) <= 0.5 else 0

    all_criteria = [accuracy, absolute_loss, squared_loss, log_loss]
```

7.1.3 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 dataset, 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):
    """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 = []
    for example in data:
        if random.random() < prob_test:
            test.append(example)
        else:
            train.append(example)
    return train, test
```

7.1.4 Importing Data From File

A dataset is typically loaded from a file. The default here is that it loaded from a CSV (comma separated values) file, although the separator can be changed. This assumes that all lines that contain the separator are valid data (so 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 index for the included columns, not the original 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=[], target_type= None, include_only=None, seed=None): #seed=12345):
        """create a dataset from a file
        separator is the character that separates the attributes
        num_train is a number specifying the first num_train tuples are training, or None
        prob_test is the probability an example should in the test set (if num_train is None)
        has_header is True if the first line of file is a header
        target_index specifies which feature is the target
        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
        target_type is either None for automatic detection of target type
             or one of "numeric", "boolean", "cartegorical"
        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:
            self.display(1,"Loading",file_name)
            # data_all = csv.reader(csvfile,delimiter=separator) # for more complicated CSV files
            data_all = (line.strip().split(separator) for line in csvfile)
            if 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,
                                      seed=seed, target_type=target_type)
```

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

learnProblem.py — (continued)

```
class Data_from_files(Data_set):
    def __init__(self, train_file_name, test_file_name, separator=',',
                 has_header=False, target_index=0, boolean_features=True,
                 categorical=[], target_type= None, include_only=None):
        """create a dataset from separate training and file
        separator is the character that separates the attributes
        num_train is a number specifying the first num_train tuples are
            training, or None
        prob_test is the probability an example should in the test set (if
            num_train is None)
        has_header is True if the first line of file is a header
        target_index specifies which feature is the target
        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
        target_type is either None for automatic detection of target type
             or one of "numeric", "boolean", "cartegorical"
        include_only is a list or set of indexes of columns to include
        """
        self.boolean_features = boolean_features
        with open(train_file_name,'r',newline='') as train_file:
          with open(test_file_name,'r',newline='') as test_file:
            # data_all = csv.reader(csvfile,delimiter=separator) # for more
                complicated CSV files
            train_data = (line.strip().split(separator) for line in
                train_file)
            test_data = (line.strip().split(separator) for line in
                test_file)
            if include_only is not None:
                train_data = ([v for (i,v) in enumerate(line) if i in
                    include_only]
                                for line in train_data)
                test_data = ([v for (i,v) in enumerate(line) if i in
                    include_only]
                                for line in test_data)
            if has_header: # this assumes the training file has a header
                and the test file doesn't
                header = next(train_data)
            else:
                header = None
            train_tuples = [interpret_elements(d) for d in train_data if
                len(d)>1]
```

```
        test_tuples = [interpret_elements(d) for d in test_data if
            len(d)>1]
        Data_set.__init__(self,train_tuples, test_tuples,
                              target_index=target_index, header=header)
```

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 numeric if possible.
    Otherwise remove initial and trailing spaces.
    """
    res = []
    for e in str_list:
        try:
            res.append(int(e))
        except ValueError:
            try:
                res.append(float(e))
            except ValueError:
                se = e.strip()
                if se in ["True","true","TRUE"]:
                    res.append(True)
                elif se in ["False","false","FALSE"]:
                    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 special cases of these are **kernel**s; mapping the original feature space into a new space, which allow a neat way to do learning in the augmented space for many mappings (the "kernel trick"). This is beyond the scope of AIPython; those interested should read about support vector machines.

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

```

```python
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)

```python
# from learnProblem import Data_set_augmented,prod_feat
# data = Data_from_file('data/holiday.csv', has_header=True, num_train=19,
    target_index=-1)
# data = Data_from_file('data/iris.data', prob_test=1/3, 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)

```python
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 more sophisticated method does not do better than this, it is not useful. This also provides the base case for some methods, such as decision-tree learning.

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

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

- a point prediction, where 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.

For regression, we do the first of these. For classification, we do the second. The third can be implemented by having multiple indicator functions for the target.

Here are some prediction functions that take in an enumeration of values, a domain, and returns a value or dictionary of $\{value : prediction\}$. Note that `cmedian` returns one of middle values when there are an even number of examples, whereas `median` gives the average of them (and so `cmedian` is applicable for ordinals that cannot be considered cardinal values). Similarly, `cmode` picks one of the values when more than one value has the maximum number of elements.

learnNoInputs.py — Learning ignoring all input features

```
from learnProblem import Evaluate
import math, random, collections, statistics
import utilities # argmax for (element,value) pairs

class Predict(object):
    """The class of prediction methods for a list of values.
    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."""

    ### The following return a distribution over values (for classification)
    def empirical(data, domain=[0,1], icount=0):
        "empirical dist "
        # returns a distribution over values
        counts = {v:icount for v in domain}
        for e in data:
            counts[e] += 1
        s = sum(counts.values())
        return {k:v/s for (k,v) in counts.items()}

    def bounded_empirical(data, domain=[0,1], bound=0.01):
        "bounded empirical"
        return {k:min(max(v,bound),1-bound) for (k,v) in
            Predict.empirical(data, domain).items()}

    def laplace(data, domain=[0,1]):
        "Laplace       "  # for categorical data
        return Predict.empirical(data, domain, icount=1)

    def cmode(data, domain=[0,1]):
        "mode          " # for categorical data
        md = statistics.mode(data)
        return {v: 1 if v==md else 0 for v in domain}

    def cmedian(data, domain=[0,1]):
        "median        " # for categorical data
        md = statistics.median_low(data) # always return one of the values
        return {v: 1 if v==md else 0 for v in domain}

    ### The following return a single prediction (for regression). domain
        is ignored.
```

```python

    def mean(data, domain=[0,1]):
        "mean          "
        # returns a real number
        return statistics.mean(data)

    def rmean(data, domain=[0,1], mean0=0, pseudo_count=1):
        "regularized mean"
        # returns a real number.
        # mean0 is the mean to be used for 0 data points
        # With mean0=0.5, pseudo_count=2, same as laplace for [0,1] data
        # this works for enumerations as well as lists
        sum = mean0 * pseudo_count
        count = pseudo_count
        for e in data:
            sum += e
            count += 1
        return sum/count

    def mode(data, domain=[0,1]):
        "mode          "
        return statistics.mode(data)

    def median(data, domain=[0,1]):
        "median        "
        return statistics.median(data)

    all = [empirical, mean, rmean, bounded_empirical, laplace, cmode, mode, median,cmedian]

    # The following suggests appropriate predictions as a function of the target type
    select = {"boolean": [empirical, bounded_empirical, laplace, cmode, cmedian],
             "categorical": [empirical, bounded_empirical, laplace, cmode, cmedian],
             "numeric": [mean, rmean, 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 generate a random number in range [0,1] and return 1 if that number is less than *prob*. A prediction is computed by applying the predictor to the training data, which is evaluated on the test set. This is repeated `num_samples` times.

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

learnNoInputs.py — (continued)

```
def test_no_inputs(error_measures = Evaluate.all_criteria,
    num_samples=10000, test_size=10 ):
   for train_size in [1,2,3,4,5,10,20,100,1000]:
      results = {predictor: {error_measure: 0 for error_measure in
          error_measures}
                     for predictor in Predict.all}
      for sample in range(num_samples):
          prob = random.random()
          training = [1 if random.random()<prob else 0 for i in
              range(train_size)]
          test = [1 if random.random()<prob else 0 for i in
              range(test_size)]
          for predictor in Predict.all:
              prediction = predictor(training)
              for error_measure in error_measures:
                  results[predictor][error_measure] += sum(
                      error_measure(prediction,actual) for actual in
                      test)/test_size
      print(f"For training size {train_size}:")
      print("  Predictor\t","\t".join(error_measure.__doc__ for
                                          error_measure in
                                              error_measures),sep="\t")
      for predictor in Predict.all:
         print(f"  {predictor.__doc__}",
                   "\t".join("{:.7f}".format(results[predictor][error_measure]/num_samples)
                                  for error_measure in
                                      error_measures),sep="\t")

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

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

(a) Squared error

(b) Absolute error

(c) Log loss

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

Exercise 7.5 Suggest some other predictions that only take the training 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, Evaluate
from learnNoInputs import Predict
import math

class DT_learner(Learner):
    def __init__(self,
                 dataset,
                 split_to_optimize=Evaluate.log_loss, # to minimize for at
                     each split
                 leaf_prediction=Predict.empirical, # what to use for value
                     at leaves
                 train=None,                  # used for cross validation
                 max_num_cuts=8, # maximum number of conditions to split a
                     numeric feature into
                 gamma=1e-7 , # minimum improvement needed to expand a node
                 min_child_weight=10):
        self.dataset = dataset
        self.target = dataset.target
        self.split_to_optimize = split_to_optimize
        self.leaf_prediction = leaf_prediction
        self.max_num_cuts = max_num_cuts
        self.gamma = gamma
        self.min_child_weight = min_child_weight
        if train is None:
            self.train = self.dataset.train
        else:
            self.train = train

    def learn(self, max_num_cuts=8):
        """learn a decision tree"""
        return self.learn_tree(self.dataset.conditions(self.max_num_cuts),
            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 only splits if the best split increases the error by at least gamma. This implies it does not split when:

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

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

learnDT.py — (continued)

```
    def learn_tree(self, conditions, data_subset):
        """returns a decision tree
        conditions 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
        """
        self.display(2,f"learn_tree with {len(conditions)} features and {len(data_subset)} examples")
        split, partn = self.select_split(conditions, data_subset)
        if split is None: # no split; return a point prediction
            prediction = self.leaf_value(data_subset, self.target.frange)
            self.display(2,f"leaf prediction for {len(data_subset)} examples is {prediction}")
            def leaf_fun(e):
                return prediction
            leaf_fun.__doc__ = str(prediction)
            leaf_fun.num_leaves = 1
            return leaf_fun
        else:  # a split succeeded
            false_examples, true_examples = partn
            rem_features = [fe for fe in conditions 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__ = (f"(if {split.__doc__} then {true_tree.__doc__}"
```

```
                    f" else {false_tree.__doc__})")
        fun.num_leaves = true_tree.num_leaves + false_tree.num_leaves
        return fun
```

learnDT.py — (continued)

```
    def leaf_value(self, egs, domain):
        return self.leaf_prediction((self.target(e) for e in egs), domain)

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

        conditions is a non-empty list of features.
        returns feature, partition
        where feature is an input feature with the smallest error as
              judged by split_to_optimize or
              feature==None if there are no splits that improve the error
        partition is a pair (false_examples, true_examples) if feature is
            not None
        """
        best_feat = None # best feature
        # best_error = float("inf") # infinity - more than any error
        best_error = self.sum_losses(data_subset) - self.gamma
        self.display(3," no split has
            error=",best_error,"with",len(conditions),"conditions")
        best_partition = None
        for feat in conditions:
            false_examples, true_examples = partition(data_subset,feat)
            if
                min(len(false_examples),len(true_examples))>=self.min_child_weight:
                err = (self.sum_losses(false_examples)
                       + self.sum_losses(true_examples))
                self.display(3," split on",feat.__doc__,"has error=",err,
                          "splits
                              into",len(true_examples),":",len(false_examples),"gamma=",self.gamma)
                if err < best_error:
                    best_feat = feat
                    best_error=err
                    best_partition = false_examples, true_examples
        self.display(2,"best split is on",best_feat.__doc__,
                              "with err=",best_error)
        return best_feat, best_partition

    def sum_losses(self, data_subset):
        """returns sum of losses for dataset (with no more splits)
        There a single prediction for all leaves using leaf_prediction
        It is evaluated using split_to_optimize
        """
        prediction = self.leaf_value(data_subset, self.target.frange)
        error = sum(self.split_to_optimize(prediction, self.target(e))
                     for e in data_subset)
```

```
        return error

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

Test cases:

learnDT.py — (continued)

```
from learnProblem import Data_set, Data_from_file

def testDT(data, print_tree=True, selections = None, **tree_args):
    """Prints errors and the trees for various evaluation criteria and ways
        to select leaves.
    """
    if selections == None: # use selections suitable for target type
        selections = Predict.select[data.target.ftype]
    evaluation_criteria = Evaluate.all_criteria
    print("Split Choice","Leaf Choice\t","#leaves",'\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,
                                 **tree_args).learn()
            print(crit.__doc__, leaf.__doc__, tree.num_leaves,
                    "\t".join("{:.7f}".format(data.evaluate_dataset(data.test,
                        tree, ecrit))
                                for ecrit in evaluation_criteria),sep="\t")
            if print_tree:
                print(tree.__doc__)

#DT_learner.max_display_level = 4
if __name__ == "__main__":
    # Choose one of the data files
    #data=Data_from_file('data/SPECT.csv', target_index=0);
        print("SPECT.csv")
    #data=Data_from_file('data/iris.data', target_index=-1);
        print("iris.data")
    data = Data_from_file('data/carbool.csv', target_index=-1, seed=123)
    #data = Data_from_file('data/mail_reading.csv', target_index=-1);
        print("mail_reading.csv")
```

```
    #data = Data_from_file('data/holiday.csv', has_header=True,
        num_train=19, target_index=-1); print("holiday.csv")
    testDT(data, print_tree=False)
```

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

Exercise 7.6 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 log loss) as the measure of improvement when building the tree, even in they want to have non-probabilistic predictions in the final tree. Does this work better than myopically choosing the split that is best for the evaluation criteria we will use to judge the final prediction?

7.5 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 dataset, 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, Evaluate
from learnNoInputs import Predict
from learnDT import DT_learner
import matplotlib.pyplot as plt
import random

class K_fold_dataset(object):
    def __init__(self, training_set, num_folds):
        self.data = training_set.train.copy()
        self.target = training_set.target
        self.input_features = training_set.input_features
        self.num_folds = num_folds
        self.conditions = training_set.conditions

        random.shuffle(self.data)
        self.fold_boundaries = [(len(self.data)*i)//num_folds
                                for i in range(0,num_folds+1)]

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

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

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

learnCrossValidation.py — (continued)

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

The *plot_error* method plots the average error as a function of a the minimum number of examples in decision-tree search, both for the validation set and for the test set. The error on the validation set can be used to tune the parameter — choose the value of the parameter that minimizes the error. The error on the test set cannot be used to tune the parameters; if is were to be used this way then it cannot be used to test.

learnCrossValidation.py — (continued)

```
def plot_error(data, criterion=Evaluate.squared_loss,
    leaf_prediction=Predict.empirical,
                num_folds=5, maxx=None, xscale='linear'):
    """Plots the error on the validation set and the test set
    with respect to settings of the minimum number of examples.
    xscale should be 'log' or 'linear'
    """
    plt.ion()
    plt.xscale(xscale) # change between log and linear scale
    plt.xlabel("min_child_weight")
    plt.ylabel("average "+criterion.__doc__)
    folded_data = K_fold_dataset(data, num_folds)
    if maxx == None:
        maxx = len(data.train)//2+1
    verrors = [] # validation errors
    terrors = [] # test set errors
    for mcw in range(1,maxx):
        verrors.append(folded_data.validation_error(DT_learner,criterion,leaf_prediction=leaf_pred
                                                min_child_weight=mcw))
        tree = DT_learner(data, criterion, leaf_prediction=leaf_prediction,
            min_child_weight=mcw).learn()
        terrors.append(data.evaluate_dataset(data.test,tree,criterion))
    plt.plot(range(1,maxx), verrors, ls='-',color='k',
            label="validation for "+criterion.__doc__)
    plt.plot(range(1,maxx), terrors, ls='--',color='k',
            label="test set for "+criterion.__doc__)
    plt.legend()
    plt.draw()

```

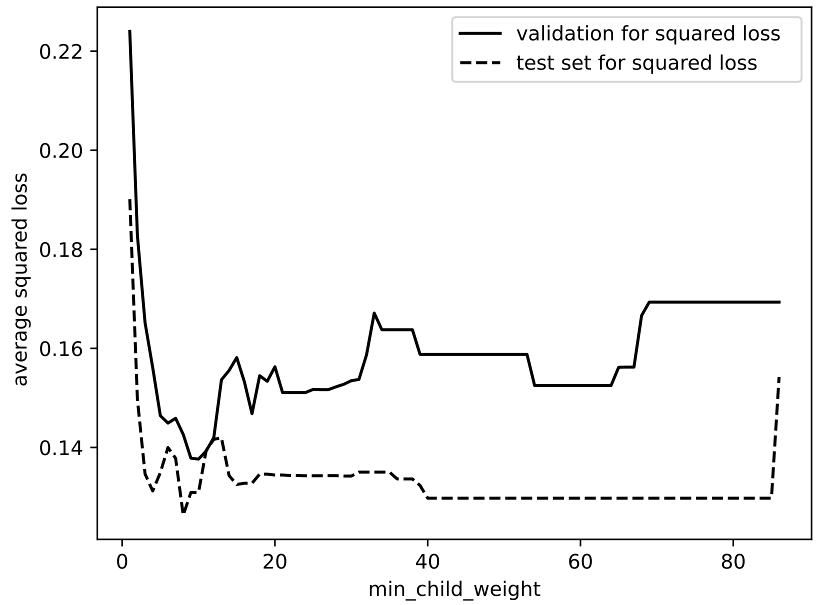


Figure 7.2: `plot_error` for SPECT dataset

```
# The following produces the graphs of Figure 7.18 of Poole and Mackworth
    [2023]
# data = Data_from_file('data/SPECT.csv',target_index=0, seed=123)
# plot_error(data, criterion=Evaluate.log_loss,
    leaf_prediction=Predict.laplace)

#also try:
# plot_error(data)
# data = Data_from_file('data/carbool.csv', target_index=-1, seed=123)
```

Figure 7.2 shows the average squared loss in the validation and test sets as a function of the `min_child_weight` in the decision-tree learning algorithm. (SPECT data with seed 12345 followed by `plot_error(data)`). Different seeds will produce different graphs. The assumption behind cross vaildation is that the parameter that minimizes the loss on the validation set, will be a good parameter for the test set.

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`, with a different seed, 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 is a stochastic gradient descent searcher for linear regression and classification.

learnLinear.py — Linear Regression and Classification

```
from learnProblem import Learner
import random, math

class Linear_learner(Learner):
    def __init__(self, dataset, train=None,
                 learning_rate=0.1, max_init = 0.2,
                 squashed=True, batch_size=10):
        """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.batch_size = batch_size
        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. Only the number of iterations is specified; the other parameters it gets from the class.

learnLinear.py — (continued)

```
    def learn(self,num_iter=100):
        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):
                error = self.predictor(e) - self.target(e)
                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
        return self.predictor
```

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

learnLinear.py — (continued)

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

$sigmoid(x)$ is the function

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

The inverse of *sigmoid* is the *logit* function

learnLinear.py — (continued)

```
def sigmoid(x):
    return 1/(1+math.exp(-x))

def logit(x):
    return -math.log(1/x-1)
```

sigmoid$([x_0, v_2, \dots])$ returns $[v_0, v_2, \dots]$ where

$$v_i = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

The inverse of *sigmoid* is the *logit* function

learnLinear.py — (continued)

```
def softmax(xs,domain=None):
    """xs is a list of values, and
    domain is the domain (a list) or None if the list should be returned
    returns a distribution over the domain (a dict)
    """
    m = max(xs) # use of m prevents overflow (and all values underflowing)
    exps = [math.exp(x-m) for x in xs]
    s = sum(exps)
    if domain:
        return {d:v/s for (d,v) in zip(domain,exps)}
    else:
        return [v/s for v in exps]

def indicator(v, domain):
    return [1 if v==dv else 0 for dv in domain]
```

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

learnLinear.py — (continued)

```
from learnProblem import Data_set, Data_from_file, Evaluate
from learnProblem import Evaluate
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 Evaluate.all_criteria:
        test_error = data.evaluate_dataset(data.test, learner.predictor,
            ecrit)
        print("   Average", ecrit.__doc__, "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=Evaluate.squared_loss,
```

```
                step=1,
                num_steps=1000,
                log_scale=True,
                legend_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

    """
    if legend_label != "": legend_label+=" "
    plt.ion()
    plt.xlabel("step")
    plt.ylabel("Average "+criterion.__doc__)
    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', has_header=True,
            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='-',label=legend_label+"training")
    plt.plot(range(1,num_steps+1,step),test_errors,ls='--',label=legend_label+"test")
    plt.legend()
    plt.draw()
    learner.display(1, "Train error:",train_errors[-1],
                         "Test error:",test_errors[-1])

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

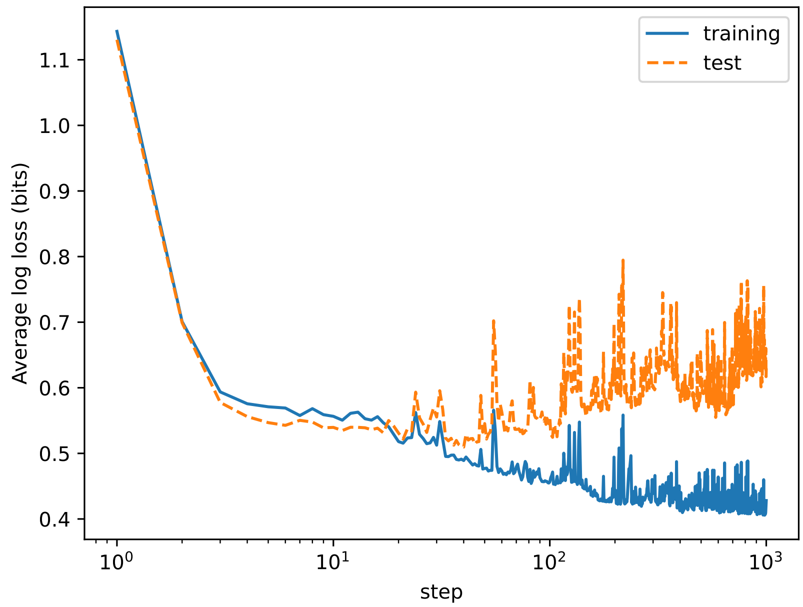


Figure 7.3: `plot_steps` for SPECT dataset

```
# 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,
    seed=123)
# dataplus = Data_set_augmented(data, [], [prod_feat])
# plot_steps(data=data, num_steps=1000)
# plot_steps(data=dataplus, num_steps=1000) # warning very slow
```

Figure 7.3 shows the result of `plot_steps(data=data, num_steps=1000)` in the code above. What would you expect to happen with the augmented data (with extra features)? Hint: think about underitting and overfitting.

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(data,
             learner = None,
             minx = 0,
             maxx = 5,
             step_size = 0.01,  # for plotting
             label = "function"):
    plt.ion()
    plt.xlabel("x")
    plt.ylabel("y")
    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,
                 Evaluate.squared_loss))
    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,
              learner_class = Linear_learner,
              max_degree = 5,
              minx = 0,
              maxx = 5,
              num_iter = 1000000,
              learning_rate = 0.00001,
              step_size = 0.01, # for plotting
              ):
    plt.ion()
```

```
    plt.xlabel("x")
    plt.ylabel("y")
    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, Evaluate.squared_loss))
        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:
# data0 = Data_from_file('data/simp_regr.csv', prob_test=0,
    boolean_features=False, target_index=-1)
# plot_prediction(data0)
# plot_polynomials(data0)
# What if the step size was bigger?
#datam = Data_from_file('data/mail_reading.csv', target_index=-1)
#plot_prediction(datam)
```

7.7 Boosting

The following code implements functional gradient boosting for regression.

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

learnBoosting.py — Functional Gradient Boosting

```
from learnProblem import Data_set, Learner, Evaluate
from learnNoInputs import Predict
from learnLinear import sigmoid
import statistics
import random
```

```

class Boosted_dataset(Data_set):
    def __init__(self, base_dataset, offset_fun, subsample=1.0):
        """new dataset which is like base_dataset,
           but offset_fun(e) is subtracted from the target of each example e
        """
        self.base_dataset = base_dataset
        self.offset_fun = offset_fun
        self.train = random.sample(base_dataset.train,int(subsample*len(base_dataset.train)))
        self.test = base_dataset.test
        #Data_set.__init__(self, base_dataset.train, base_dataset.test,
        #                  base_dataset.prob_test, base_dataset.target_index)

        #def create_features(self):
        """creates new features - called at end of Data_set.init()
        defines a new target
        """
        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
        newout.ftype = self.infer_type(newout.frange)
        self.target = newout

    def conditions(self, *args, colsample_bytree=0.5, **nargs):
        conds = self.base_dataset.conditions(*args, **nargs)
        return random.sample(conds, int(colsample_bytree*len(conds)))
```

A boosting learner takes in a dataset and a base learner, and returns a new predictor. The base learner, takes a dataset, and returns a Learner object.

learnBoosting.py — (continued)

```
class Boosting_learner(Learner):
    def __init__(self, dataset, base_learner_class, subsample=0.8):
        self.dataset = dataset
        self.base_learner_class = base_learner_class
        self.subsample = subsample
        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] # list of base learners
        self.predictors = [self.predictor] # list of predictors
        self.errors = [data.evaluate_dataset(data.test, self.predictor, Evaluate.squared_loss)]
        self.display(1,"Predict mean test set mean squared loss=", self.errors[0] )


```

```
    def learn(self, num_ensembles=10):
        """adds num_ensemble learners to the ensemble.
        returns a new predictor.
        """
        for i in range(num_ensembles):
            train_subset = Boosted_dataset(self.dataset, self.predictor, subsample=self.subsample)
            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.predictors.append(new_pred)
            self.errors.append(data.evaluate_dataset(data.test, self.predictor, Evaluate.squared_loss))
            self.display(1,f"Iteration {len(self.offsets)-1},treesize = {new_offset.num_leaves}. mean squared loss={self.errors[-1]}")
        return self.predictor
```

For testing, *sp_DT_learner* returns a learner that predicts the mean at the leaves and is evaluated using squared loss. It can also take arguments to change the default arguments for the trees.

learnBoosting.py — (continued)

```
# Testing

from learnDT import DT_learner
from learnProblem import Data_set, Data_from_file

def sp_DT_learner(split_to_optimize=Evaluate.squared_loss,
                             leaf_prediction=Predict.mean,**nargs):
    """Creates a learner with different default arguments replaced by **nargs
    """
    def new_learner(dataset):
        return DT_learner(dataset,split_to_optimize=split_to_optimize,
                                 leaf_prediction=leaf_prediction, **nargs)
    return new_learner

#data = Data_from_file('data/car.csv', target_index=-1) regression
data = Data_from_file('data/student/student-mat-nq.csv', separator=';',has_header=True,target_index=-1,seed=13,include_only=list(range(30))+[32]) #2.0537973790924946
#data = Data_from_file('data/SPECT.csv', target_index=0, seed=62) #123)
#data = Data_from_file('data/mail_reading.csv', target_index=-1)
#data = Data_from_file('data/holiday.csv', has_header=True, num_train=19, target_index=-1)
#learner10 = Boosting_learner(data, sp_DT_learner(split_to_optimize=Evaluate.squared_loss,
```

```
    leaf_prediction=Predict.mean, min_child_weight=10))
#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_trees(data, steps=10, mcws=[30,20,20,10], gammas=
    [100,200,300,500]):
    # to reduce clutter uncomment one of following two lines
    #mcws=[10]
    #gammas=[200]
    learners = [(mcw, gamma, Boosting_learner(data,
        sp_DT_learner(min_child_weight=mcw, gamma=gamma)))
                  for gamma in gammas for mcw in mcws
                  ]
    plt.ion()
    plt.xscale('linear') # change between log and linear scale
    plt.xlabel("number of trees")
    plt.ylabel("mean squared loss")
    markers = (m+c for c in ['k','g','r','b','m','c','y'] for m in
        ['-','--','-.',':'])
    for (mcw,gamma,learner) in learners:
        data.display(1,f"min_child_weight={mcw}, gamma={gamma}")
        learner.learn(steps)
        plt.plot(range(steps+1), learner.errors, next(markers),
                    label=f"min_child_weight={mcw}, gamma={gamma}")
    plt.legend()
    plt.draw()

# plot_boosting_trees(data)
```

7.7.1 Gradient Tree Boosting

The following implements gradient Boosted trees for classification. If you want to use this gradient tree boosting for a real problem, we recommend using **XGBoost** [Chen and Guestrin, 2016] or **LightGBM** [Ke, Meng, Finley, Wang, Chen, Ma, Ye, and Liu, 2017].

`GTB_learner` subclasses `DT-learner`. The method `learn_tree` is used unchanged. `DT-learner` assumes that the value at the leaf is the prediction of the leaf, thus `leaf_value` needs to be overridden. It also assumes that all nodes at a leaf have the same prediction, but in GBT the elements of a leaf can have different values, depending on the previous trees. Thus `sum_losses` also needs to be overridden.

learnBoosting.py — (continued)

```
class GTB_learner(DT_learner):
    def __init__(self, dataset, number_trees, lambda_reg=1, gamma=0,
        **dtargs):
```

```
        DT_learner.__init__(self, dataset,
            split_to_optimize=Evaluate.log_loss, **dtargs)
        self.number_trees = number_trees
        self.lambda_reg = lambda_reg
        self.gamma = gamma
        self.trees = []

    def learn(self):
        for i in range(self.number_trees):
            tree =
                self.learn_tree(self.dataset.conditions(self.max_num_cuts),
                self.train)
            self.trees.append(tree)
            self.display(1,f"""Iteration {i} treesize = {tree.num_leaves}
                train logloss={
               self.dataset.evaluate_dataset(self.dataset.train,
                   self.gtb_predictor, Evaluate.log_loss)
                   } test logloss={
               self.dataset.evaluate_dataset(self.dataset.test,
                   self.gtb_predictor, Evaluate.log_loss)}""")
        return self.gtb_predictor

    def gtb_predictor(self, example, extra=0):
        """prediction for example,
        extras is an extra contribution for this example being considered
        """
        return sigmoid(sum(t(example) for t in self.trees)+extra)

    def leaf_value(self, egs, domain=[0,1]):
        """value at the leaves for examples egs
        domain argument is ignored"""
        pred_acts = [(self.gtb_predictor(e),self.target(e)) for e in egs]
        return sum(a-p for (p,a) in pred_acts) /(sum(p*(1-p) for (p,a) in
            pred_acts)+self.lambda_reg)


    def sum_losses(self, data_subset):
        """returns sum of losses for dataset (assuming a leaf is formed
            with no more splits)
        """
        leaf_val = self.leaf_value(data_subset)
        error = sum(Evaluate.log_loss(self.gtb_predictor(e,leaf_val),
            self.target(e))
                     for e in data_subset) + self.gamma
        return error
```

Testing

learnBoosting.py — (continued)

```
# data = Data_from_file('data/carbool.csv', target_index=-1, seed=123)
# gtb_learner = GTB_learner(data, 10)
```

```
# gtb_learner.learn()
```

Chapter 8

Neural Networks and Deep Learning

Warning: this is not meant to be an efficient implementation of deep learning. If you want to do serious machine learning on meduim-sized or large data, we recommend Keras (`https://keras.io`) [Chollet, 2021] or PyTorch (`https://pytorch.org`), which are very efficient, particularly on GPUs. They are, however, black boxes. The AIPython neural network code should be seen like a car engine made of glass; you can see exactly how it works, even if it is not fast.

The parameters that are the same as in Keras have the same names.

8.1 Layers

A neural network is built from layers.

This provides a modular implementation of layers. Layers can easily be stacked in many configurations. A layer needs to implement a function to compute the output values from the inputs, a way to back-propagate the error, and perhaps update its parameters.

learnNN.py — Neural Network Learning

```
from learnProblem import Learner, Data_set, Data_from_file,
    Data_from_files, Evaluate
from learnLinear import sigmoid, one, softmax, indicator
import random, math, time

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 layer 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, training=False):
        """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.
        It can act differently when training and when predicting.
        """
        raise NotImplementedError("output_values") # abstract method

    def backprop(self,errors):
        """Backpropagate the errors on the outputs
        errors is a list of errors for the outputs (of length
            self.num_outputs).
        Returns 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,
          which can remember information information required for the
              back-propagation.
        """
        raise NotImplementedError("backprop") # abstract method

    def update(self):
        """updates parameters after a batch.
        overridden by layers that have parameters
        """
        pass
```

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 end of the inputs. The default initialization is the Glorot uniform initializer [Glorot and Bengio, 2010], which is the default in Keras. An alternative is to provide a limit, in which case the values are selected uniformly in the range $[-limit, limit]$. Keras treats the bias separately, and defaults to zero.

learnNN.py — (continued)

```
class Linear_complete_layer(Layer):
    """a completely connected layer"""
    def __init__(self, nn, num_outputs, limit=None):
        """A completely connected linear layer.
```

```
        nn is a neural network that the inputs come from
        num_outputs is the number of outputs
        the random initialization of parameters is in range [-limit,limit]
        """
        Layer.__init__(self, nn, num_outputs)
        if limit is None:
            limit =math.sqrt(6/(self.num_inputs+self.num_outputs))
        # self.weights[o][i] is the weight between input i and output o
        self.weights = [[random.uniform(-limit, limit) if inf < self.num_inputs else 0
                          for inf in range(self.num_inputs+1)]
                        for outf in range(self.num_outputs)]
        self.delta = [[0 for inf in range(self.num_inputs+1)]
                        for outf in range(self.num_outputs)]

    def output_values(self,input_values, training=False):
        """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 loops over the outputs.
        The bias is the *last* value of each list in self.weights.
        """
        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.delta[out][inp] += self.inputs[inp] * errors[out]
        return input_errors[:-1] # remove the error for the "1"

    def update(self):
        """updates parameters after a batch"""
        batch_step_size = self.nn.learning_rate / self.nn.batch_size
        for out in range(self.num_outputs):
            for inp in range(self.num_inputs+1):
                self.weights[out][inp] -= batch_step_size * self.delta[out][inp]
                self.delta[out][inp] = 0
```

The standard activation function for hidden nodes is the **ReLU**.

learnNN.py — (continued)

```
class ReLU_layer(Layer):
```

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

    def output_values(self, input_values, training=False):
        """Returns the outputs for the input values.
        It remembers the input values for the backprop.
        """
        self.input_values = input_values
        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)]
```

One of the old standards for the activation function for hidden layers is the sigmoid. It is included here to experiment with.

learnNN.py — (continued)

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

    def output_values(self, input_values, training=False):
        """Returns the outputs for the input values.
        It remembers the output values for the backprop.
        """
        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)]
```

8.2 Feedforward Networks

learnNN.py — (continued)

```
class NN(Learner):
    def __init__(self, dataset, validation_proportion = 0.1,
        learning_rate=0.001):
        """Creates a neural network for a dataset,
```

```
        layers is the list of layers
        """
        self.dataset = dataset
        self.output_type = dataset.target.ftype
        self.learning_rate = learning_rate
        self.input_features = dataset.input_features
        self.num_outputs = len(self.input_features)
        validation_num = int(len(self.dataset.train)*validation_proportion)
        if validation_num > 0:
            random.shuffle(self.dataset.train)
            self.validation_set = self.dataset.train[-validation_num:]
            self.training_set = self.dataset.train[:-validation_num]
        else:
            self.validation_set = []
            self.training_set = self.dataset.train
        self.layers = []
        self.bn = 0 # number of batches run

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

    def predictor(self,ex):
        """Predicts the value of the first output for example ex.
        """
        values = [f(ex) for f in self.input_features]
        for layer in self.layers:
            values = layer.output_values(values)
        return sigmoid(values[0]) if self.output_type =="boolean" \
               else softmax(values, self.dataset.target.frange) if self.output_type == "categorical" \
               else values[0]

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

The *learn* method learns a network.

learnNN.py — (continued)

```
    def learn(self, epochs=5, batch_size=32, num_iter = None, report_each=10):
        """Learns parameters for a neural network using stochastic gradient decent.
        epochs is the number of times through the data (on average)
        batch_size is the maximum size of each batch
        num_iter is the number of iterations over the batches
            - overrides epochs if provided (allows for fractions of epochs)
```

```
        report_each means give the errors after each multiple of that iterations
        """
        self.batch_size = min(batch_size, len(self.training_set)) # don't have batches bigger than training size
        if num_iter is None:
             num_iter = (epochs * len(self.training_set)) // self.batch_size
        #self.display(0,"Batch\t","\t".join(criterion.__doc__ for criterion in Evaluate.all_criteria))
        for i in range(num_iter):
            batch = random.sample(self.training_set, self.batch_size)
            for e in batch:
                # compute all outputs
                values = [f(e) for f in self.input_features]
                for layer in self.layers:
                    values = layer.output_values(values, training=True)
                # backpropagate
                predicted = [sigmoid(v) for v in values] if self.output_type == "boolean"\
                             else softmax(values) if self.output_type == "categorical"\
                             else values
                actuals = indicator(self.dataset.target(e), self.dataset.target.frange) \
                            if self.output_type == "categorical"\
                            else [self.dataset.target(e)]
                errors = [pred-obsd for (obsd,pred) in zip(actuals,predicted)]
                for layer in reversed(self.layers):
                    errors = layer.backprop(errors)
            # Update all parameters in batch
            for layer in self.layers:
                layer.update()
            self.bn+=1
            if (i+1)%report_each==0:
                self.display(0,self.bn,"\t",
                            "\t\t".join("{:.4f}".format(
                                self.dataset.evaluate_dataset(self.validation_set, self.predictor, criterion))
                              for criterion in Evaluate.all_criteria), sep="")
```

8.3 Improved Optimization

8.3.1 Momentum

learnNN.py — (continued)

```
class Linear_complete_layer_momentum(Linear_complete_layer):
    """a completely connected layer"""
```

```
    def __init__(self, nn, num_outputs, limit=None, alpha=0.9, epsilon =
        1e-07, vel0=0):
        """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
        vel0 is the initial velocity for each parameter
        """
        Linear_complete_layer.__init__(self, nn, num_outputs, limit=limit)
        # self.weights[o][i] is the weight between input i and output o
        self.velocity = [[vel0 for inf in range(self.num_inputs+1)]
                        for outf in range(self.num_outputs)]
        self.alpha = alpha
        self.epsilon = epsilon

    def update(self):
        """updates parameters after a batch"""
        batch_step_size = self.nn.learning_rate / self.nn.batch_size
        for out in range(self.num_outputs):
            for inp in range(self.num_inputs+1):
                self.velocity[out][inp] = self.alpha*self.velocity[out][inp]
                    - batch_step_size * self.delta[out][inp]
                self.weights[out][inp] += self.velocity[out][inp]
                self.delta[out][inp] = 0
```

8.3.2 RMS-Prop

learnNN.py — (continued)

```
class Linear_complete_layer_RMS_Prop(Linear_complete_layer):
    """a completely connected layer"""
    def __init__(self, nn, num_outputs, limit=None, rho=0.9, epsilon =
        1e-07):
        """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
        """
        Linear_complete_layer.__init__(self, nn, num_outputs, limit=limit)
        # self.weights[o][i] is the weight between input i and output o
        self.ms = [[0 for inf in range(self.num_inputs+1)]
                        for outf in range(self.num_outputs)]
        self.rho = rho
        self.epsilon = epsilon

    def update(self):
        """updates parameters after a batch"""
        for out in range(self.num_outputs):
            for inp in range(self.num_inputs+1):
```

```
                gradient = self.delta[out][inp] / self.nn.batch_size
                self.ms[out][inp] = self.rho*self.ms[out][inp]+ (1-self.rho)
                    * gradient**2
                self.weights[out][inp] -=
                    self.nn.learning_rate/(self.ms[out][inp]+self.epsilon)**0.5
                    * gradient
                self.delta[out][inp] = 0
```

8.4 Dropout

Dropout is implemented as a layer.

learnNN.py — (continued)

```
from utilities import flip
class Dropout_layer(Layer):
    """Dropout layer
    """

    def __init__(self, nn, rate=0):
        """
        rate is fraction of the input units to drop. 0 =< rate < 1
        """
        self.rate = rate
        Layer.__init__(self, nn)

    def output_values(self, input_values, training=False):
        """Returns the outputs for the input values.
        It remembers the input values for the backprop.
        """
        if training:
            scaling = 1/(1-self.rate)
            self.mask = [0 if flip(self.rate) else 1
                           for _ in input_values]
            return [x*y*scaling for (x,y) in zip(input_values, self.mask)]
        else:
            return input_values

    def backprop(self,errors):
        """Returns the derivative of the errors"""
        return [x*y for (x,y) in zip(errors, self.mask)]

class Dropout_layer_0(Layer):
    """Dropout layer
    """

    def __init__(self, nn, rate=0):
        """
        rate is fraction of the input units to drop. 0 =< rate < 1
        """
        self.rate = rate
```

```
        Layer.__init__(self, nn)

    def output_values(self, input_values, training=False):
        """Returns the outputs for the input values.
        It remembers the input values for the backprop.
        """
        if training:
            scaling = 1/(1-self.rate)
            self.outputs= [0 if flip(self.rate) else inp*scaling # make 0
                with probability rate
                           for inp in input_values]
            return self.outputs
        else:
            return input_values

    def backprop(self,errors):
        """Returns the derivative of the errors"""
        return errors
```

8.4.1 Examples

The following constructs a neural network with one hidden layer. The output is assumed to be Boolean or Real. If it is categorical, the final layer should have the same number of outputs as the number of cetegories (so it can use a softmax).

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=0.3, target_index=0,
    seed=12345)
#data = Data_from_file('data/iris.data', prob_test=0.2, target_index=-1) #
    150 examples approx 120 test + 30 test
#data = Data_from_file('data/if_x_then_y_else_z.csv', num_train=8,
    target_index=-1) # not linearly sep
#data = Data_from_file('data/holiday.csv', target_index=-1) #,
    num_train=19)
#data = Data_from_file('data/processed.cleveland.data', target_index=-1)
#random.seed(None)

# nn3 is has a single hidden layer of width 3
nn3 = NN(data, validation_proportion = 0)
nn3.add_layer(Linear_complete_layer(nn3,3))
#nn3.add_layer(Sigmoid_layer(nn3))
nn3.add_layer(ReLU_layer(nn3))
nn3.add_layer(Linear_complete_layer(nn3,1)) # when using
    output_type="boolean"
#nn3.learn(epochs = 100)

```

```
# nn3do is like nn3 but with dropout on the hidden layer
nn3do = NN(data, validation_proportion = 0)
nn3do.add_layer(Linear_complete_layer(nn3do,3))
#nn3.add_layer(Sigmoid_layer(nn3)) # comment this or the next
nn3do.add_layer(ReLU_layer(nn3do))
nn3do.add_layer(Dropout_layer(nn3do, rate=0.5))
nn3do.add_layer(Linear_complete_layer(nn3do,1))
#nn3do.learn(epochs = 100)

# nn3_rmsp is like nn3 but uses RMS prop
nn3_rmsp = NN(data, validation_proportion = 0)
nn3_rmsp.add_layer(Linear_complete_layer_RMS_Prop(nn3_rmsp,3))
#nn3_rmsp.add_layer(Sigmoid_layer(nn3_rmsp)) # comment this or the next
nn3_rmsp.add_layer(ReLU_layer(nn3_rmsp))
nn3_rmsp.add_layer(Linear_complete_layer_RMS_Prop(nn3_rmsp,1))
#nn3_rmsp.learn(epochs = 100)

# nn3_m is like nn3 but uses momentum
mm1_m = NN(data, validation_proportion = 0)
mm1_m.add_layer(Linear_complete_layer_momentum(mm1_m,3))
#mm1_m.add_layer(Sigmoid_layer(mm1_m)) # comment this or the next
mm1_m.add_layer(ReLU_layer(mm1_m))
mm1_m.add_layer(Linear_complete_layer_momentum(mm1_m,1))
#mm1_m.learn(epochs = 100)

# nn2 has a single a hidden layer of width 2
nn2 = NN(data, validation_proportion = 0)
nn2.add_layer(Linear_complete_layer_RMS_Prop(nn2,2))
nn2.add_layer(ReLU_layer(nn2))
nn2.add_layer(Linear_complete_layer_RMS_Prop(nn2,1))

# nn5 is has a single hidden layer of width 5
nn5 = NN(data, validation_proportion = 0)
nn5.add_layer(Linear_complete_layer_RMS_Prop(nn5,5))
nn5.add_layer(ReLU_layer(nn5))
nn5.add_layer(Linear_complete_layer_RMS_Prop(nn5,1))

# nn0 has no hidden layers, and so is just logistic regression:
nn0 = NN(data, validation_proportion = 0) #learning_rate=0.05)
nn0.add_layer(Linear_complete_layer(nn0,1))
# Or try this for RMS-Prop:
#nn0.add_layer(Linear_complete_layer_RMS_Prop(nn0,1))
```

Plotting. Figure 8.1 shows the training and test performance on the SPECT dataset for the architectures above. Note the nn5 test has infinite log loss after about 45,000 steps. The noisyness of the predictions might indicate that the step size is too big. This was produced by the code below:

learnNN.py — (continued)

```
from learnLinear import plot_steps
from learnProblem import Evaluate
```

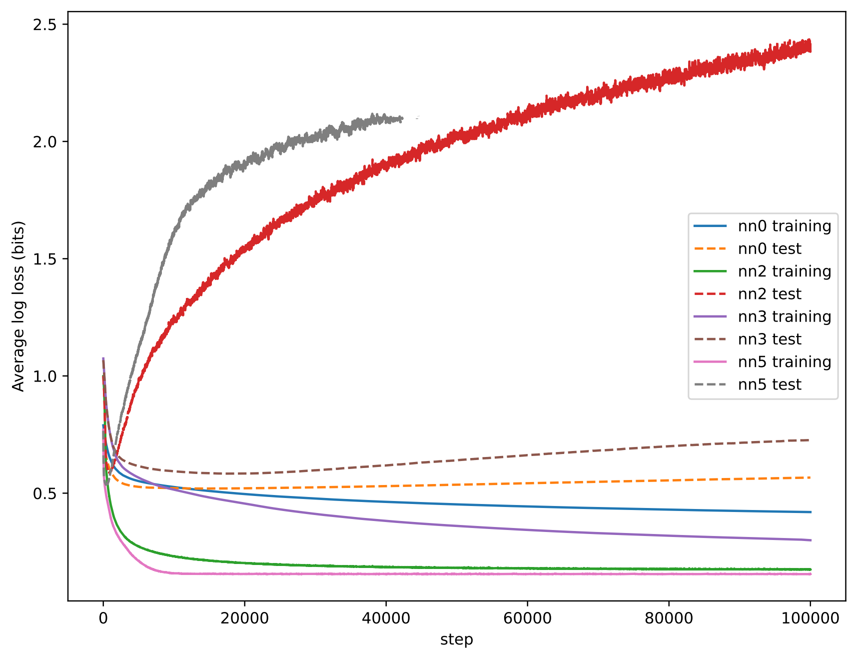


Figure 8.1: Plotting train and test log loss for various algorithms on SPECT dataset

```

# To show plots first choose a criterion to use
# crit = Evaluate.log_loss
# crit = Evaluate.accuracy
# plot_steps(learner = nn0, data = data, criterion=crit, num_steps=10000,
    log_scale=False, legend_label="nn0")
# plot_steps(learner = nn2, data = data, criterion=crit, num_steps=10000,
    log_scale=False, legend_label="nn2")
# plot_steps(learner = nn3, data = data, criterion=crit, num_steps=10000,
    log_scale=False, legend_label="nn3")
# plot_steps(learner = nn5, data = data, criterion=crit, num_steps=10000,
    log_scale=False, legend_label="nn5")

# for (nn,nname) in [(nn0,"nn0"),(nn2,"nn2"),(nn3,"nn3"),(nn5,"nn5")]:
    plot_steps(learner = nn, data = data, criterion=crit,
    num_steps=100000, log_scale=False, legend_label=nname)

# Print some training examples
#for eg in random.sample(data.train,10): print(eg,nn3.predictor(eg))

# Print some test examples
#for eg in random.sample(data.test,10): print(eg,nn3.predictor(eg))
```

```

# To see the weights learned in linear layers
# nn3.layers[0].weights
# nn3.layers[2].weights

# Print test:
# for e in data.train: print(e,nn0.predictor(e))

def test(data, hidden_widths = [5], epochs=100,
            optimizers = [Linear_complete_layer,
                      Linear_complete_layer_momentum,
                          Linear_complete_layer_RMS_Prop]):
    data.display(0,"Batch\t","\t".join(criterion.__doc__ for criterion in
        Evaluate.all_criteria))
    for optimizer in optimizers:
        nn = NN(data)
        for width in hidden_widths:
            nn.add_layer(optimizer(nn,width))
            nn.add_layer(ReLU_layer(nn))
        if data.target.ftype == "boolean":
            nn.add_layer(optimizer(nn,1))
        else:
            error(f"Not implemented: {data.output_type}")
        nn.learn(epochs)
```

The following tests on MNIST. The original files are from `http://yann.lecun.com/exdb/mnist/`. This code assumes you use the csv files from `https://pjreddie.com/projects/mnist-in-csv/`, and put them in the directory `../MNIST/`. Note that this is **very** inefficient; you would be better to use Keras or Pytorch. There are $28 * 28 = 784$ input units and 512 hidden units, which makes 401,408 parameters for the lowest linear layer. So don't be surprised when it takes many hours in AIPython (even if it only takes a few seconds in Keras).

learnNN.py — (continued)

```
# Simplified version: (6000 training instances)
# data_mnist = Data_from_file('../MNIST/mnist_train.csv', prob_test=0.9,
    target_index=0, boolean_features=False, target_type="categorical")

# Full version:
# data_mnist = Data_from_files('../MNIST/mnist_train.csv',
    '../MNIST/mnist_test.csv', target_index=0, boolean_features=False,
    target_type="categorical")

# nn_mnist = NN(data_mnist, validation_proportion = 0.02,
    learning_rate=0.001) #validation set = 1200
# nn_mnist.add_layer(Linear_complete_layer_RMS_Prop(nn_mnist,512));
    nn_mnist.add_layer(ReLU_layer(nn_mnist));
    nn_mnist.add_layer(Linear_complete_layer_RMS_Prop(nn_mnist,10))
# start_time = time.perf_counter();nn_mnist.learn(epochs=1,
    batch_size=128);end_time = time.perf_counter();print("Time:", end_time
```

```
    - start_time,"seconds") #1 epoch
# determine test error:
# data_mnist.evaluate_dataset(data_mnist.test, nn_mnist.predictor,
    Evaluate.accuracy)
# Print some random predictions:
# for eg in random.sample(data_mnist.test,10):
    print(data_mnist.target(eg),nn_mnist.predictor(eg),nn_mnist.predictor(eg)[data_mnist.target(eg
```

Exercise 8.1 In the definition of *nn*3 above, for each of the following, first hypothesize what will happen, then test your hypothesis, then explain whether you testing confirms your hypothesis or not. Test it for more than one data set, and use more than one run for each data set.

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

(b) Which is faster, 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 a ReLU layer then a sigmoid layer after the first linear layer and before the second linear layer?

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

Exercise 8.2 Do some

Chapter 9

Reasoning with Uncertainty

9.1 Representing Probabilistic Models

A probabilisitic model uses the same definition of a variable as a CSP (Section 4.1.1, page 57). A variable consists of a name, a domain and an optional (x,y) position (for displaying). The domain of a variable is a list or a tuple, as the ordering will matter in the representation of factors.

9.2 Representing Factors

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

A variable assignment, or just **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 # 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, given={}):
        """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 = [v for v in self.variables if v not in given]
        else: #enforce ordering and allow for extra variables in ordering
            variables = [v for v in variables if v in self.variables and v
                not in given]
        head = "\t".join(str(v) for v in variables)
        return head+"\n"+self.ass_to_str(variables, given, 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"+"{:.6f}".format(self.get_value(asst)) )

```

```
    __repr__ = __str__
```

9.3 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__
```

A constant CPD has no parents, and has probability 1 when the variable has the value specified, and 0 when the variable has a different value.

probFactors.py — (continued)

```
class ConstantCPD(CPD):
    def __init__(self, variable, value):
        CPD.__init__(self, variable, [])
        self.value = value
    def get_value(self, assignment):
        return 1 if self.value==assignment[self.child] else 0
```

9.3.1 Logistic Regression

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

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

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

probFactors.py — (continued)

```
from learnLinear import sigmoid, logit

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)
```

9.3.2 Noisy-or

A **noisy-or**, for Boolean variable X with Boolean parents $Y_1 \dots Y_k$ is parametrized by $k+1$ parameters $p_0, p_1, \dots, p_k$, where each $0 \le p_i \le 1$. The sematics is defined as though there are $k+1$ hidden variables $Z_0, Z_1 \dots Z_k$, where $P(Z_0) = p_0$ and $P(Z_i \mid Y_i) = p_i$ for $i \ge 1$, and where X is true if and only if $Z_0 \vee Z_1 \vee \dots \vee Z_k$ (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
```

9.3.3 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 $V_1, V_2, \ldots, V_k$, the value of $f(V_1 = v_1, V_2 = v_1, \ldots, V_k = v_k)$ is stored in $f[v_1][v_2]\ldots[v_k]$.

If the domain of V_i is $[0, \ldots, n_i - 1]$ 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
```

9.4 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, fontsize=10, facecolor='orange'):
        plt.ion()  # interactive
        ax = plt.figure().gca()
        ax.set_axis_off()
        plt.title(self.title, fontsize=fontsize)
        bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5",facecolor=facecolor)
```

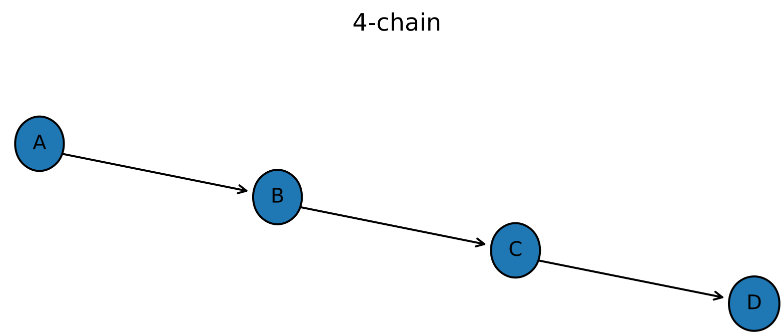


Figure 9.1: bn_4ch.show()

```
        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', fontsize=fontsize)
            else:
                x,y = var.position
                plt.text(x,y,var.name,bbox=bbox,ha='center',
                    fontsize=fontsize)
```

9.4.1 Example Belief Networks

A Chain of 4 Variables

The first example belief network is a simple chain $A \longrightarrow B \longrightarrow C \longrightarrow D$, shown in Figure 9.1.

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

probGraphicalModels.py — (continued)

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

boolean = [False, True]
A = Variable("A", boolean, position=(0,0.8))
B = Variable("B", boolean, position=(0.333,0.7))
C = Variable("C", boolean, position=(0.666,0.6))
D = Variable("D", boolean, position=(1,0.5))

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

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

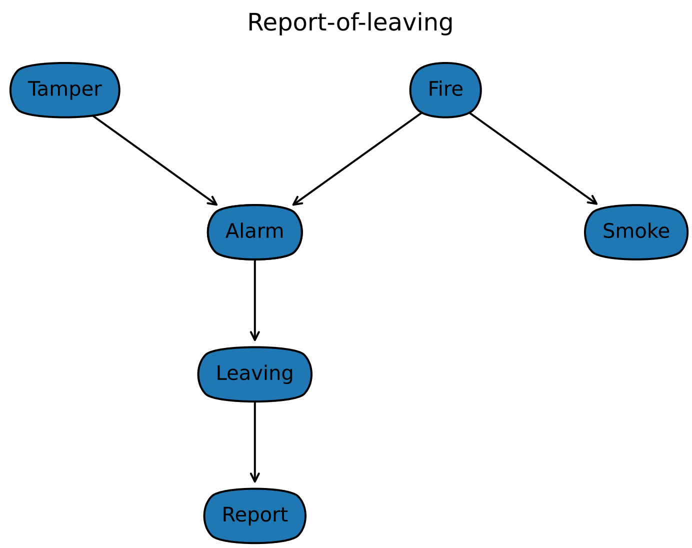


Figure 9.2: The report-of-leaving belief network

Report-of-Leaving Example

The second belief network, `bn_report`, is Example 9.13 of Poole and Mackworth [2023] (`http://artint.info`). The output of `bn_report.show()` is shown in Figure 9.2 of this document.

probGraphicalModels.py — (continued)

```
# Belief network report-of-leaving example (Example 9.13 shown in Figure 9.3) of
# Poole and Mackworth, Artificial Intelligence, 2023 http://artint.info

Alarm =  Variable("Alarm", boolean, position=(0.366,0.5))
Fire =   Variable("Fire",  boolean, position=(0.633,0.75))
Leaving = Variable("Leaving", boolean, position=(0.366,0.25))
Report = Variable("Report", boolean, position=(0.366,0.0))
Smoke =  Variable("Smoke", boolean, position=(0.9,0.5))
Tamper = Variable("Tamper", boolean, position=(0.1,0.75))

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]])
```

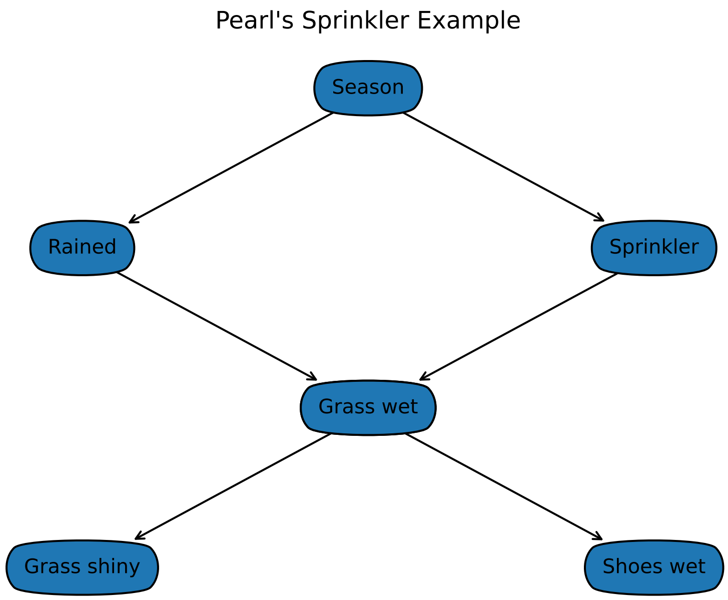


Figure 9.3: The sprinkler belief network

```

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 9.3 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))
Shoes_wet = Variable("Shoes wet", boolean, position=(0.9,0))

f_season = Prob(Season,[],{'summer':0.5, 'winter':0.5})
f_sprinkler = Prob(Sprinkler,[Season],{'summer':{'on':0.9,'off':0.1},
                                   'winter':{'on':0.01,'off':0.99}})
f_rained = Prob(Rained,[Season],{'summer':[0.9,0.1], 'winter': [0.2,0.8]})
f_wet = Prob(Grass_wet,[Sprinkler,Rained], {'on': [[0.1,0.9],[0.01,0.99]],
                                       'off':[[0.99,0.01],[0.3,0.7]]})
```

```
f_shiny = Prob(Grass_shiny, [Grass_wet], [[0.95,0.05], [0.3,0.7]])
f_shoes = Prob(Shoes_wet, [Grass_wet], [[0.98,0.02], [0.35,0.65]])

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})

bn_sprinkler_soff = BeliefNetwork("Pearl's Sprinkler Example
    (do(Sprinkler=off))",
                      {Season, Sprinkler, Rained, Grass_wet, Grass_shiny,
                          Shoes_wet},
                      {f_season, f_rained, f_wet, f_shiny, f_shoes,
                          Prob(Sprinkler,[],{'on':0,'off':1})})
```

Bipartite Diagnostic Model with Noisy-or

The belief network bn_no1 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 symptoms and the symptoms are only connected to the diseases. The output of bn_no1.show() is shown in Figure 9.4 of this document.

probGraphicalModels.py — (continued)

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

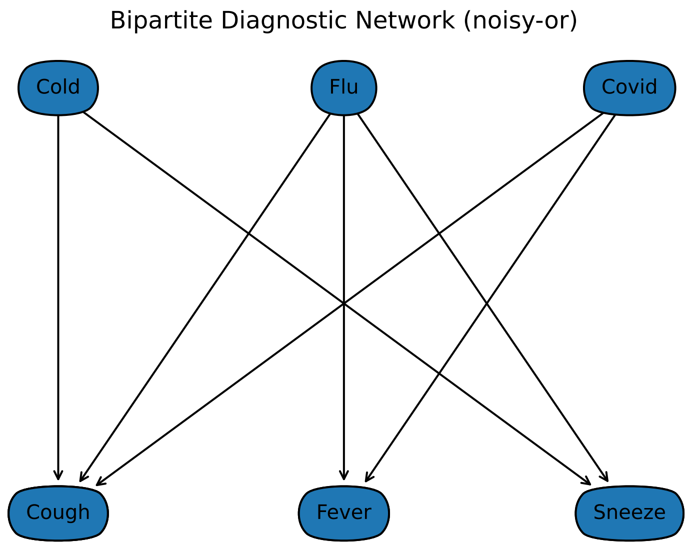


Figure 9.4: A bipartite diagnostic network

```
# example from box "Noisy-or compared to logistic regression"
# X = Variable("X",boolean)
# w0 = 0.01
# print(NoisyOR(X,[A,B,C,D],[w0, 1-(1-0.05)/(1-w0), 1-(1-0.1)/(1-w0),
    1-(1-0.2)/(1-w0), 1-(1-0.2)/(1-w0), ]).to_table(given={X:True}))
```

Bipartite Diagnostic Model with Logistic Regression

The belief network `bn_lr1` is a bipartite diagnostic model, with independent diseases, and the 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 9.4). 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())

# example from box "Noisy-or compared to logistic regression"
# from learnLinear import sigmoid, logit
# w0=logit(0.01)
# X = Variable("X",boolean)
# print(LogisticRegression(X,[A,B,C,D],[w0, logit(0.05)-w0, logit(0.1)-w0,
    logit(0.2)-w0, logit(0.2)-w0]).to_table(given={X:True}))
# try to predict what would happen (and then test) if we had
# w0=logit(0.01)
```

9.5 Inference Methods

Each of the inference methods implements the query method that computes the posterior probability of a variable given a dictionary of $\{variable : value\}$ observations. The methods are Displayable because they implement the *display* method which is 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 __init__(self,gm=None):
        self.gm = gm

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

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

probGraphicalModels.py — (continued)

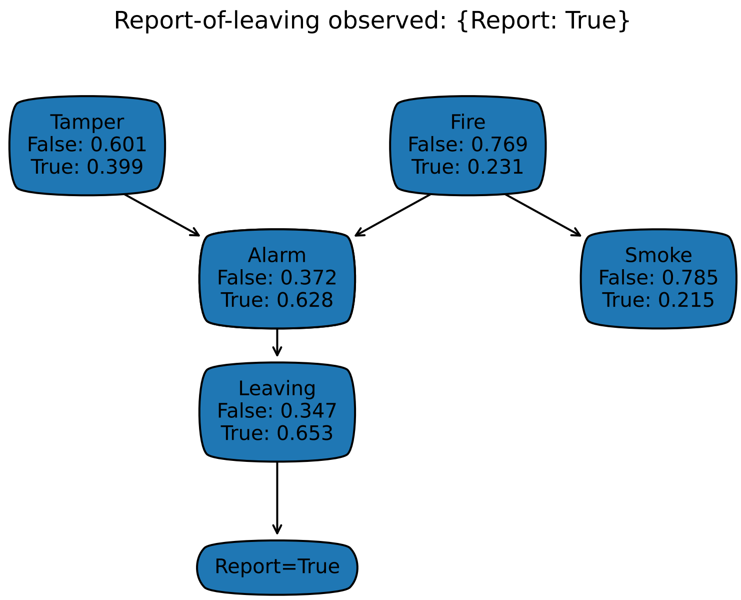


Figure 9.5: The report-of-leaving belief network with posterior distributions

```
    def testIM(self, threshold=0.0000000001):
        solver = self(bn_4ch)
        res = solver.query(B,{D:True})
        correct_answer = 0.429632380245
        assert correct_answer-threshold < res[True] <
            correct_answer+threshold, \
               f"value {res[True]} not in desired range for
                   {self.method_name}"
        print(f"Unit test passed for {self.method_name}.")
```

The following draws the posterior distribution of all variables. Figure 9.5 shows the result of `bn_reportRC.show_post({Report:True})` when run after loading `probRC.py` (see below).

probGraphicalModels.py — (continued)

```
    def show_post(self, obs={}, num_format="{:.3f}", fontsize=10,
        facecolor='orange'):
        """draws the graphical model conditioned on observations obs
          num_format is number format (allows for more or less precision)
          fontsize gives size of the text
          facecolor gives the color of the nodes
        """
        plt.ion()  # interactive
        ax = plt.figure().gca()
        ax.set_axis_off()
```

```
        plt.title(self.gm.title+" observed: "+str(obs), fontsize=fontsize)
        bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
            facecolor=facecolor)
        for var in reversed(self.gm.topological_sort()):
            distn = self.query(var, obs=obs)
            if var in obs:
                text = var.name + "=" + str(obs[var])
            else:
                text = var.name + "\n" + "\n".join(str(d)+":
                    "+num_format.format(v) for (d,v) in distn.items())
            if self.gm.var2parents[var]:
                for par in self.gm.var2parents[var]:
                    ax.annotate(text, par.position, xytext=var.position,
                                arrowprops={'arrowstyle':'<-'},bbox=bbox,
                                ha='center', fontsize=fontsize)
            else:
                x,y = var.position
                plt.text(x,y,text, bbox=bbox, ha='center', fontsize=fontsize)
```

9.6 Naive Search

An instance of a *ProbSearch* object takes in a graphical model. The query method uses naive search to compute the probability of a query variable given observations on other variables. See Figure 9.9 of Poole and Mackworth [2023].

probRC.py — Recursive Conditioning for Graphical Models

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

class ProbSearch(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):
        InferenceMethod.__init__(self, gm)
        ## self.max_display_level = 3

    def query(self, qvar, obs={}, split_order=None):
        """computes P(qvar | obs) where
        qvar is the query variable
        obs is a variable:value dictionary
        split_order is a list of the non-observed non-query variables in gm
        """
        if qvar in obs:
```

```
            return {val:(1 if val == obs[qvar] else 0) for val in
                qvar.domain}
        else:
            if split_order == None:
                split_order = [v for v in self.gm.variables if (v not in
                    obs) and v != qvar]
            unnorm = [self.prob_search({qvar:val}|obs, self.gm.factors,
                split_order)
                          for val in qvar.domain]
            p_obs = sum(unnorm)
            return {val:pr/p_obs for val,pr in zip(qvar.domain, unnorm)}
```

The following is the naive search-based algorithm. It is exponential in the number of variables, so is not very useful. However, it is simple, and useful to understand before looking at the more complicated algorithm used in the subclass.

probRC.py — (continued)

```
    def prob_search(self, context, factors, split_order):
        """simple search algorithm
        context is a variable:value dictionary
        factors is a set of factors
        split_order is a list of variables in factors not assigned in
            context
        returns sum over variable assignments to variables in split order
            or product of factors """
        self.display(2,"calling prob_search,",(context,factors))
        if not factors:
            return 1
        elif to_eval := {fac for fac in factors if
            fac.can_evaluate(context)}:
            # evaluate factors when all variables are assigned
            self.display(3,"prob_search evaluating factors",to_eval)
            val = math.prod(fac.get_value(context) for fac in to_eval)
            return val * self.prob_search(context, factors-to_eval,
                split_order)
        else:
            total = 0
            var = split_order[0]
            self.display(3, "prob_search branching on", var)
            for val in var.domain:
                total += self.prob_search({var:val}|context, factors,
                    split_order[1:])
            self.display(3, "prob_search branching on", var,"returning",
                total)
            return total
```

9.7 Recursive Conditioning

The **recursive conditioning** algorithm adds forgetting and caching and recognizing disconnected components to the naive searcg. We do this by adding a cache and redefining the recursive search algorithm. In inherits the query method. See Figure 9.12 of Poole and Mackworth [2023].

probRC.py — (continued)

```
class ProbRC(ProbSearch):
    def __init__(self,gm=None):
        self.cache = {(frozenset(), frozenset()):1}
        ProbSearch.__init__(self,gm)

    def prob_search(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 assigned
            in context
        returns sum over variable assignments to variables in split_order
                   of the product of factors
        """
        self.display(3,"calling rc,",(context,factors))
        ce = (frozenset(context.items()), frozenset(factors)) # key for the
            cache entry
        if ce in self.cache:
            self.display(3,"rc cache lookup",(context,factors))
            return self.cache[ce]
#        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.prob_search({key:val for (key,val) in
                context.items()
                                 if key not in vars_not_in_factors},
                             factors, split_order)
        elif to_eval := {fac for fac in factors if
            fac.can_evaluate(context)}:
            # evaluate factors when all variables are assigned
            self.display(3,"rc evaluating factors",to_eval)
            val = math.prod(fac.get_value(context) for fac in to_eval)
            if val == 0:
                return 0
            else:
             return val * self.prob_search(context, {fac for fac in factors
```

```
                                                 if fac not in to_eval}, split_order)
        elif len(comp := connected_components(context, factors, split_order)) > 1:
            # there are disconnected components
            self.display(3,"splitting into connected components",comp,"in context",context)
            return(math.prod(self.prob_search(context,f,eo) for (f,eo) in comp))
        else:
            assert split_order, "split_order should not be empty to get here"
            total = 0
            var = split_order[0]
            self.display(3, "rc branching on", var)
            for val in var.domain:
                total += self.prob_search({var:val}|context, factors, split_order[1:])
            self.cache[ce] = total
            self.display(2, "rc branching on", var,"returning", total)
            return total
```

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

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

probRC.py — (continued)

```
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 not in
                                                                component_variables])
                                                                )
    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 = ProbRC(bn_4ch)
## bn_4chv.query(A,{})
## bn_4chv.query(D,{})
## InferenceMethod.max_display_level = 3 # show more detail in displaying
## InferenceMethod.max_display_level = 1 # show less detail in displaying
## bn_4chv.query(A,{D:True},[C,B])
## bn_4chv.query(B,{A:True,D:False})

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

## To display resulting posteriors try:
# bn_reportRC.show_post({})
# bn_reportRC.show_post({Smoke:False})
# bn_reportRC.show_post({Report:True})
# bn_reportRC.show_post({Report:True, Smoke:False})
```

```

## Note what happens to the cache when these are called in turn:
## bn_reportRC.query(Tamper,{Report:True},
    split_order=[Smoke,Fire,Alarm,Leaving])
## bn_reportRC.query(Smoke,{Report:True},
    split_order=[Tamper,Fire,Alarm,Leaving])

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

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

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

9.8 Variable Elimination

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

probVE.py — Variable Elimination for Graphical Models

```
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):
```

```
        InferenceMethod.__init__(self, 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, Shoes_wet
```

```
bn_sprinklerv = VE(bn_sprinkler)
## bn_sprinklerv.query(Shoes_wet,{})
## bn_sprinklerv.query(Shoes_wet,{Rained:True})
## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:True})
## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:False,Rained:True})

from 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)
```

9.9 Stochastic Simulation

9.9.1 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 9.1

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)
```

9.9.2 Sampling Methods for Belief Network Inference

A *SamplingInferenceMethod* is an *InferenceMethod*, but the query method also takes arguments for the number of samples and the sample-order (which is an ordering of factors). The first methods assume a belief network (and not an undirected graphical model).

probStochSim.py — (continued)

```
class SamplingInferenceMethod(InferenceMethod):
    """The abstract class of sampling-based belief network inference
        methods"""

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

    def query(self,qvar,obs={},number_samples=1000,sample_order=None):
        raise NotImplementedError("SamplingInferenceMethod query") #
            abstract
```

9.9.3 Rejection Sampling

probStochSim.py — (continued)

```
class RejectionSampling(SamplingInferenceMethod):
    """The class that queries Graphical Models using Rejection Sampling.

    gm is a belief network to query
    """
    method_name = "rejection sampling"

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

    def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
        """computes P(qvar | obs) where
        qvar is a variable.
        obs is a {variable:value} dictionary.
        sample_order is a list of variables where the parents
          come before the variable.
        """
        if sample_order is None:
            sample_order = self.gm.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.gm.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 distribution we also include raw counts
        dist = {c:v/tot if tot>0 else 1/len(qvar.domain) for (c,v) in
            counts.items()}
        dist["raw_counts"] = counts
        return dist
```

9.9.4 Likelihood Weighting

Likelihood weighting includes a weight for each sample. Instead of rejecting samples based on observations, likelihood weighting changes the weights of the sample in proportion with the probability of the observation. The weight then becomes the probability that the variable would have been rejected.

probStochSim.py — (continued)

```
class LikelihoodWeighting(SamplingInferenceMethod):
    """The class that queries Graphical Models using Likelihood weighting.

    gm is a belief network to query
    """
    method_name = "likelihood weighting"

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

    def query(self,qvar,obs={},number_samples=1000,sample_order=None):
        """computes P(qvar | obs) where
        qvar is a variable.
        obs is a {variable:value} dictionary.
        sample_order is a list of factors where factors defining the parents
          come before the factors for the child.
        """
        if sample_order is None:
```

```
            sample_order = self.gm.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.gm.var2cpt[nvar]
                if nvar in obs:
                    sample[nvar] = obs[nvar]
                    weight *= fac.get_value(sample)
                else:
                    val = sample_one({v:fac.get_value({**sample,nvar:v}) for
                        v in nvar.domain})
                    self.display(2,val,end="\t")
                    sample[nvar] = val
            counts[sample[qvar]] += weight
            self.display(2,weight)
        tot = sum(counts.values())
        # as well as the distribution we also include the raw counts
        dist = {c:v/tot for (c,v) in counts.items()}
        dist["raw_counts"] = counts
        return dist
```

Exercise 9.2 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).

9.9.5 Particle Filtering

In this implementation, a particle is a $\{variable : value\}$ dictionary. Because adding a new value to dictionary involves a side effect, the dictionaries need to be copied during resampling.

probStochSim.py — (continued)

```
class ParticleFiltering(SamplingInferenceMethod):
    """The class that queries Graphical Models using Particle Filtering.

    gm is a belief network to query
    """
    method_name = "particle filtering"

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

    def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
        """computes P(qvar | obs) where
```

```
        qvar is a variable.
        obs is a {variable:value} dictionary.
        sample_order is a list of factors where factors defining the parents
          come before the factors for the child.
        """
        if sample_order is None:
            sample_order = self.gm.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.gm.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_samples)]
            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
        dist = {c:v/tot for (c,v) in counts.items()}
        dist["raw_counts"] = counts
        return dist
```

Resampling

Resample is based on *sample_multiple* but works with an array of particles. (Aside: Python doesn't let us use *sample_multiple* directly as it uses a dictionary, and particles, represented as dictionaries can't be the key of dictionaries).

probStochSim.py — (continued)

```
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
```

9.9.6 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
    likelihood weighting
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(ParticleFiltering, threshold=0.1)
```

Exercise 9.3 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.

9.9.7 Gibbs Sampling

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

probStochSim.py — (continued)

```
#import random
#from probGraphicalModels import InferenceMethod

#from probStochSim import sample_one, SamplingInferenceMethod

class GibbsSampling(SamplingInferenceMethod):
    """The class that queries Graphical Models using Gibbs Sampling.

    bn is a graphical model (e.g., a belief network) to query
    """
    method_name = "Gibbs sampling"

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

    def query(self, qvar, obs={}, number_samples=1000, burn_in=100,
         sample_order=None):
        """computes P(qvar | obs) where
        qvar is a variable.
        obs is a {variable:value} dictionary.
        sample_order is a list of non-observed variables in order, or
        if sample_order None, 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.gm.variables if v not in obs]
```

```
        var_to_factors = {v:set() for v in self.gm.variables}
        for fac in self.gm.factors:
            for var in fac.variables:
                var_to_factors[var].add(fac)
        sample = {var:random.choice(var.domain) for var in variables}
        self.display(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
        dist = {c:v/tot for (c,v) in counts.items()}
        dist["raw_counts"] = counts
        return dist

#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 9.4 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 9.5 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?

9.9.8 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.items())})"
```

```python
    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 ProbRC
# Try following (but it takes a while..)
methods = [ProbRC,RejectionSampling,LikelihoodWeighting,ParticleFiltering,GibbsSampling]
#plot_mult(methods,bn_report,Tamper,True,{Report:True,Smoke:False},number_samples=100, number_runs=1000)
# plot_mult(methods,bn_report,Tamper,True,{Report:False,Smoke:True},number_samples=100, number_runs=1000)

# Sprinkler Example:
# plot_stats(bn_sprinklerr,Shoes_wet,True,{Grass_shiny:True,Rained:True},number_samples=1000)
# plot_stats(bn_sprinklerL,Shoes_wet,True,{Grass_shiny:True,Rained:True},number_samples=1000)
```

9.10 Hidden Markov Models

This code for hidden Markov models is independent of the graphical models code, to keep it simple. Section 9.11 gives code that models hidden Markov models, and more generally, dynamic belief networks, using the graphical models code.

This HMM code assumes there are multiple Boolean observation variables that depend on the current state and are independent of each other given the state.

probHMM.py — Hidden Markov Model

```
import random
from probStochSim import sample_one, sample_multiple

class HMM(object):
    def __init__(self, states, obsvars, pobs, trans, indist):
        """A hidden Markov model.
        states - set of states
        obsvars - set of observation variables
        pobs - probability of observations, pobs[i][s] is P(Obs_i=True |
            State=s)
        trans - transition probability - trans[i][j] gives P(State=j |
            State=i)
        indist - initial distribution - indist[s] is P(State_0 = s)
        """
        self.states = states
        self.obsvars = obsvars
        self.pobs = pobs
        self.trans = trans
        self.indist = indist
```

Consider the following example. Suppose you want to unobtrusively keep track of an animal in a triangular enclosure using sound. Suppose you have 3 microphones that provide unreliable (noisy) binary information at each time step. The animal is either close to one of the 3 points of the triangle or in the middle of the triangle.

probHMM.py — (continued)

```
# state
#       0=middle, 1,2,3 are corners
states1 = {'middle', 'c1', 'c2', 'c3'} # states
obs1 = {'m1','m2','m3'} # microphones
```

The observation model is as follows. If the animal is in a corner, it will be detected by the microphone at that corner with probability 0.6, and will be independently detected by each of the other microphones with a probability of 0.1. If the animal is in the middle, it will be detected by each microphone with a probability of 0.4.

probHMM.py — (continued)

```
# pobs gives the observation model:
#pobs[mi][state] is P(mi=on | state)
closeMic=0.6; farMic=0.1; midMic=0.4
pobs1 = {'m1':{'middle':midMic, 'c1':closeMic, 'c2':farMic, 'c3':farMic}, # mic 1
         'm2':{'middle':midMic, 'c1':farMic, 'c2':closeMic, 'c3':farMic}, # mic 2
         'm3':{'middle':midMic, 'c1':farMic, 'c2':farMic, 'c3':closeMic}} # mic 3
```

The transition model is as follows: If the animal is in a corner it stays in the same corner with probability 0.80, goes to the middle with probability 0.1 or goes to one of the other corners with probability 0.05 each. If it is in the middle, it stays in the middle with probability 0.7, otherwise it moves to one the corners, each with probability 0.1.

probHMM.py — (continued)

```
# trans specifies the dynamics
# trans[i] is the distribution over states resulting from state i
# trans[i][j] gives P(S=j | S=i)
sm=0.7; mmc=0.1            # transition probabilities when in middle
sc=0.8; mcm=0.1; mcc=0.05   # transition probabilities when in a corner
trans1 = {'middle':{'middle':sm, 'c1':mmc, 'c2':mmc, 'c3':mmc}, # was in middle
          'c1':{'middle':mcm, 'c1':sc, 'c2':mcc, 'c3':mcc}, # was in corner 1
          'c2':{'middle':mcm, 'c1':mcc, 'c2':sc, 'c3':mcc}, # was in corner 2
          'c3':{'middle':mcm, 'c1':mcc, 'c2':mcc, 'c3':sc}} # was in corner 3
```

Initially the animal is in one of the four states, with equal probability.

probHMM.py — (continued)

```
# initially we have a uniform distribution over the animal's state
indist1 = {st:1.0/len(states1) for st in states1}

hmm1 = HMM(states1, obs1, pobs1, trans1, indist1)
```

9.10.1 Exact Filtering for HMMs

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

probHMM.py — (continued)

```
from display import Displayable

class HMMVEfilter(Displayable):
    def __init__(self,hmm):
        self.hmm = hmm
```

```
        self.state_dist = hmm.indist

    def filter(self, obsseq):
        """updates and returns the state distribution following the
            sequence of
        observations in obsseq using variable elimination.

        Note that it first advances time.
        This is what is required if it is called sequentially.
        If that is not what is wanted initially, do an observe first.
        """
        for obs in obsseq:
            self.advance()    # advance time
            self.observe(obs) # observe
        return self.state_dist

    def observe(self, obs):
        """updates state conditioned on observations.
        obs is a list of values for each observation variable"""
        for i in self.hmm.obsvars:
            self.state_dist = {st:self.state_dist[st]*(self.hmm.pobs[i][st]
                                                  if obs[i] else
                                                      (1-self.hmm.pobs[i][st]))
                                   for st in self.hmm.states}
        norm = sum(self.state_dist.values()) # normalizing constant
        self.state_dist = {st:self.state_dist[st]/norm for st in
            self.hmm.states}
        self.display(2,"After observing",obs,"state
            distribution:",self.state_dist)

    def advance(self):
        """advance to the next time"""
        nextstate = {st:0.0 for st in self.hmm.states}  # distribution over
            next states
        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
        self.display(2,"After advancing state
            distribution:",self.state_dist)
```

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':1, 'm2':0, 'm3':1}])

# 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 9.6 The representation assumes that there are a list of Boolean observations. Extend the representation so that the each observation variable can have multiple discrete values. You need to choose a representation for the model, and change the algorithm.

9.10.2 Localization

The localization example in the book is a controlled HMM, where there is a given action at each time and the transition depends on the action. In this class, the transition is set to None initially, and needs to be provided with an action to determine the transition probability.

probLocalization.py — Controlled HMM and Localization example

```python
from probHMM import HMMVEfilter, HMM
from display import Displayable
import matplotlib.pyplot as plt
from matplotlib.widgets import Button, CheckButtons

class HMM_Controlled(HMM):
    """A controlled HMM, where the transition probability depends on the
        action.
       Instead of the transition probability, it has a function act2trans
       from action to transition probability.
       Any algorithms need to select the transition probability according
           to the action.
    """
    def __init__(self, states, obsvars, pobs, act2trans, indist):
        self.act2trans = act2trans
        HMM.__init__(self, states, obsvars, pobs, None, indist)


local_states = list(range(16))
door_positions = {2,4,7,11}
```

```
def prob_door(loc): return 0.8 if loc in door_positions else 0.1
local_obs = {'door':[prob_door(i) for i in range(16)]}
act2trans = {'right': [[0.1 if next == current
                              else 0.8 if next == (current+1)%16
                              else 0.074 if next == (current+2)%16
                              else 0.002 for next in range(16)] for current in range(16)],
                      'left': [[0.1 if next == current
                              else 0.8 if next == (current-1)%16
                              else 0.074 if next == (current-2)%16
                              else 0.002 for next in range(16)] for current in range(16)]}
hmm_16pos = HMM_Controlled(local_states, {'door'}, local_obs, act2trans, [1/16 for i in range(16)])
```

To change the VE localization code to allow for controlled HMMs, notice that the action selects which transition probability to us.

probLocalization.py — (continued)

```
class HMM_Local(HMMVEfilter):
    """VE filter for controlled HMMs
    """
    def __init__(self, hmm):
        HMMVEfilter.__init__(self, hmm)

    def go(self, action):
        self.hmm.trans = self.hmm.act2trans[action]
        self.advance()

loc_filt = HMM_Local(hmm_16pos)
# loc_filt.observe({'door':True}); loc_filt.go("right"); loc_filt.observe({'door':False}); loc_filt.go("right"); loc_filt.observe({'door':True})
# loc_filt.state_dist
```

The following lets us interactively move the agent and provide observations. It shows the distribution over locations.

probLocalization.py — (continued)

```
class Show_Localization(Displayable):
    def __init__(self,hmm):
        self.hmm = hmm
        self.loc_filt = HMM_Local(hmm)
        fig,(self.ax) = plt.subplots()
        plt.subplots_adjust(bottom=0.2)
        left_butt = Button(plt.axes([0.05,0.02,0.1,0.05]), "left")
        left_butt.on_clicked(self.left)
        right_butt = Button(plt.axes([0.25,0.02,0.1,0.05]), "right")
        right_butt.on_clicked(self.right)
        door_butt = Button(plt.axes([0.45,0.02,0.1,0.05]), "door")
        door_butt.on_clicked(self.door)
```

```
        nodoor_butt = Button(plt.axes([0.65,0.02,0.1,0.05]), "no door")
        nodoor_butt.on_clicked(self.nodoor)
        reset_butt = Button(plt.axes([0.85,0.02,0.1,0.05]), "reset")
        reset_butt.on_clicked(self.reset)
               #this makes sure y-axis goes to 1, graph overwritten in draw_dist
        self.draw_dist()
        plt.show()

    def draw_dist(self):
        self.ax.clear()
        plt.ylim(0,1)
        self.ax.set_ylabel("Probability")
        self.ax.set_xlabel("Location")
        self.ax.set_title("Location Probability Distribution")
        self.ax.set_xticks(self.hmm.states)
        vals = [self.loc_filt.state_dist[i] for i in self.hmm.states]
        self.bars = self.ax.bar(self.hmm.states, vals, color='black')
        self.ax.bar_label(self.bars,["{v:.2f}".format(v=v) for v in vals],
            padding = 1)
        plt.draw()

    def left(self,event):
        self.loc_filt.go("left")
        self.draw_dist()
    def right(self,event):
        self.loc_filt.go("right")
        self.draw_dist()
    def door(self,event):
        self.loc_filt.observe({'door':True})
        self.draw_dist()
    def nodoor(self,event):
        self.loc_filt.observe({'door':False})
        self.draw_dist()
    def reset(self,event):
        self.loc_filt.state_dist = {i:1/16 for i in range(16)}
        self.draw_dist()

# sl = Show_Localization(hmm_16pos)
```

9.10.3 Particle Filtering for HMMs

In this implementation a particle is just a state. If you want to do some form of smoothing, a particle should probably be a history of states. This maintains, *particles*, an array of states, *weights* an array of (non-negative) real numbers, such that *weights*[*i*] is the weight of *particles*[*i*].

probHMM.py — (continued)

```
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':1, 'm2':0, 'm3':1}])
```

Exercise 9.7 A form of importance sampling can be obtained by not resampling. Is it better or worse than particle filtering? Hint: you need to think about how they can be compared. Is the comparison different if there are more states than particles?

Exercise 9.8 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.

9.10.4 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)
```

9.11 Dynamic Belief Networks

A **dynamic belief network (DBN)** is a belief network that extends in time.

There are a number of ways that reasoning can be carried out in a DBN, including:

- Rolling out the DBN for some time period, and using standard belief network inference. The latest time that needs to be in the rolled out network is the time of the latest observation or the time of a query (whichever is later). This allows us to observe any variables at any time and query any variables at any time. This is covered in Section 9.11.2.

- An unrolled belief network may be very large, and we might only be interested in asking about "now". In this case we can just representing the variables "now". In this approach we can observe and query the current variables. We can them move to the next time. This does not allow for arbitrary historical queries (about the past or the future), but can be much simpler. This is covered in Section 9.11.3.

9.11.1 Representing Dynamic Belief Networks

To specify a DBN, 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 variable import Variable
from probGraphicalModels import GraphicalModel, BeliefNetwork
from probFactors import Prob, Factor, CPD
from probVE import VE
from display import Displayable

class DBNvariable(Variable):
    """A random variable that incorporates the stage (time)

    A 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])

dbn1 = 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])
```

9.11.2 Unrolling DBNs

probDBN.py — (continued)

```
class BNfromDBN(BeliefNetwork):
    """Belief Network unrolled from a dynamic belief network
    """

    def __init__(self,dbn,horizon):
        """dbn is the dynamic belief network being unrolled
        horizon>0 is the number of steps (so there will be horizon+1 variables for each DBN variable.
        """
        self.name2var = {var.basename: [DBNvariable(var.basename,var.domain,index) for index in range(horizon+1)]
                         for var in dbn.vars_now}
        self.display(1,f"name2var={self.name2var}")
        variables = {v for vs in self.name2var.values() for v in vs}
        self.display(1,f"variables={variables}")
        bnfactors = {CPDrename(fac,{self.name2var[var.basename][0]:var
                                    for var in fac.variables})
                     for fac in dbn.init_factors}
        bnfactors |= {CPDrename(fac,{self.name2var[var.basename][i]:var
                                     for var in fac.variables if var.index=='prev'}
                               | {self.name2var[var.basename][i+1]:var
                                     for var in fac.variables if var.index=='now'})
                     for fac in dbn.transition_factors
                         for i in range(horizon)}
        self.display(1,f"bnfactors={bnfactors}")
        BeliefNetwork.__init__(self, dbn.title, variables, bnfactors)
```

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 ProbRC
#bn = BNfromDBN(dbn1,2) # construct belief network
```

```
#drc = ProbRC(bn)           # initialize recursive conditioning
#B2 = bn.name2var['B'][2]
#drc.query(B2) #P(B2)
#drc.query(bn.name2var['B'][1],{bn.name2var['B'][0]:True,bn.name2var['C'][1]:False})
    #P(B1|B0,C1)
```

9.11.3 DBN Filtering

If we only wanted to ask questions about the current state, we can save space by forgetting the history variables.

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,self.c

    def advance(self):
        """advance to the next time"""
        prev_factors = [self.make_previous(fac) for fac in
            self.current_factors]
        prev_obs = {var.previous:val for var,val in
            self.current_obs.items()}
        two_stage_factors = prev_factors + self.dbn.transition_factors
        self.current_factors =
            self.elim_vars(two_stage_factors,self.dbn.vars_prev,prev_obs)
        self.current_obs = {}

    def make_previous(self,fac):
         """Creates new factor from fac where the current variables in fac
         are renamed to previous variables.
         """
         return FactorRename(fac, {var.previous:var for var in
             fac.variables})
```

```
    def elim_vars(self,factors, vars, obs):
        for var in vars:
            if var in obs:
                factors = [self.project_observations(fac,obs) for fac in
                    factors]
            else:
                factors = self.eliminate_var(factors, var)
        return factors
```

Example queries:

probDBN.py — (continued)

```
#df = DBNVEfilter(dbn1)
#df.observe({B1:True}); df.advance(); df.observe({C1:False})
#df.query(B1)  #P(B1|B0,C1)
#df.advance(); df.query(B1)
#dfa = DBNVEfilter(dbn_an)
# dfa.observe({Mic1_1:0, Mic2_1:1, Mic3_1:1})
# dfa.advance()
# dfa.observe({Mic1_1:1, Mic2_1:0, Mic3_1:1})
# dfa.query(Pos_1)
```

Chapter 10

Learning with Uncertainty

10.1 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

```python
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

Causality

11.1 Do Questions

A causal model can answer "do" questions.

The following adds the `queryDo` method to the `InferenceMethod` class, so it can be used with any inference method.

probDo.py — Probabilistic inference with the do operator

```
from probGraphicalModels import InferenceMethod, BeliefNetwork
from probFactors import CPD, ConstantCPD

def queryDo(self, qvar, obs={}, do={}):
    assert isinstance(self.gm, BeliefNetwork), "Do only applies to belief
        networks"
    if do=={}:
        return self.query(qvar, obs)
    else:
        newfacs = ({f for (ch,f) in self.gm.var2cpt.items() if ch not in
            do} |
                   {ConstantCPD(v,c) for (v,c) in do.items()})
        self.modBN = BeliefNetwork(self.gm.title+"(mod)",
            self.gm.variables, newfacs)
        oldBN, self.gm = self.gm, self.modBN
        result = self.query(qvar, obs)
        self.gm = oldBN # restore original
        return result

InferenceMethod.queryDo = queryDo
```

probDo.py — (continued)

```
from probRC import ProbRC
```

```
from probGraphicalModels import bn_sprinkler, Season, Sprinkler, Rained,
    Grass_wet, Grass_shiny, Shoes_wet, bn_sprinkler_soff
bn_sprinklerv = ProbRC(bn_sprinkler)
## bn_sprinklerv.queryDo(Shoes_wet)
## bn_sprinklerv.queryDo(Shoes_wet,obs={Sprinkler:"off"})
## bn_sprinklerv.queryDo(Shoes_wet,do={Sprinkler:"off"})
## ProbRC(bn_sprinkler_soff).query(Shoes_wet) # should be same as previous
    case
## bn_sprinklerv.queryDo(Season, obs={Sprinkler:"off"})
## bn_sprinklerv.queryDo(Season, do={Sprinkler:"off"})
```

The following is a representation of a possible model where marijuana is a gateway drug to harder drugs (or not). Try the queries at the end.

probDo.py — (continued)

```
from probVariables import Variable
from probFactors import Prob
from probGraphicalModels import boolean, BeliefNetwork

Takes_Marijuana = Variable("Takes_Marijuana", boolean, position=(0.1,0.5))
Drug_Prone = Variable("Drug_Prone", boolean, position=(0.1,0.5)) #
    (0.5,0.9))
Side_Effects = Variable("Side_Effects", boolean, position=(0.1,0.5)) #
    (0.5,0.1))
Takes_Hard_Drugs = Variable("Takes_Hard_Drugs", boolean,
    position=(0.9,0.5))

p_tm = Prob(Takes_Marijuana, [Drug_Prone], [[0.98, 0.02], [0.2, 0.8]])
p_dp = Prob(Drug_Prone, [], [0.8, 0.2])
p_be = Prob(Side_Effects, [Takes_Marijuana], [[1, 0], [0.4, 0.6]])
p_thd = Prob(Takes_Hard_Drugs, [Side_Effects, Drug_Prone],
               # Drug_Prone=False  Drug_Prone=True
               [[[0.999, 0.001],   [0.6, 0.4]], # Side_Effects=False
                [[0.99999, 0.00001], [0.995, 0.005]]]) # Side_Effects=True

drugs = BeliefNetwork("Gateway Drug?",
                  [Takes_Marijuana,Drug_Prone,Side_Effects,Takes_Hard_Drugs],
                  [p_tm, p_dp, p_be, p_thd])

drugsq = ProbRC(drugs)
# drugsq.queryDo(Takes_Hard_Drugs)
# drugsq.queryDo(Takes_Hard_Drugs, obs = {Takes_Marijuana: True})
# drugsq.queryDo(Takes_Hard_Drugs, obs = {Takes_Marijuana: False})
# drugsq.queryDo(Takes_Hard_Drugs, do = {Takes_Marijuana: True})
# drugsq.queryDo(Takes_Hard_Drugs, do = {Takes_Marijuana: False})
```

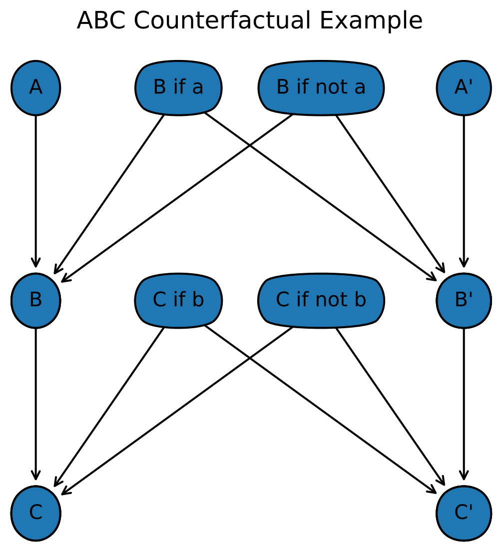


Figure 11.1: $A \rightarrow B \rightarrow C$ belief network for "what if A"

11.2 Counterfactual Example

This is for a chain $A \rightarrow B \rightarrow C$ where the query is $A{=}true, C{=}true$ is observed; what is the probability of C is A were false. See Figure 11.1.

probCounterfactual.py — Counterfactual Query Example

```
from probVariables import Variable
from probFactors import Prob
from probGraphicalModels import boolean, BeliefNetwork
from probRC import ProbRC
from probDo import queryDo

# without a deterministic system
Ap = Variable("Ap", boolean, position=(0.2,0.8))
Bp = Variable("Bp", boolean, position=(0.2,0.4))
Cp = Variable("Cp", boolean, position=(0.2,0.0))
p_Ap = Prob(Ap, [], [0.5,0.5])
p_Bp = Prob(Bp, [Ap], [[0.6,0.4], [0.6,0.4]]) # does not depend on A!
p_Cp = Prob(Cp, [Bp], [[0.2,0.8], [0.9,0.1]])
abcSimple = BeliefNetwork("ABC Simple",
                          [Ap,Bp,Cp],
                          [p_Ap, p_Bp, p_Cp])
ABCsimpq = ProbRC(abcSimple)
# ABCsimpq.show_post(obs = {Ap:True, Cp:True})
```

```

# as a deterministic system with independent noise
A = Variable("A", boolean, position=(0.2,0.8))
B = Variable("B", boolean, position=(0.2,0.4))
C = Variable("C", boolean, position=(0.2,0.0))
Aprime = Variable("A'", boolean, position=(0.8,0.8))
Bprime = Variable("B'", boolean, position=(0.8,0.4))
Cprime = Variable("C'", boolean, position=(0.8,0.0))
BifA = Variable("B if a", boolean, position=(0.4,0.8))
BifnA = Variable("B if not a", boolean, position=(0.6,0.8))
CifB = Variable("C if b", boolean, position=(0.4,0.4))
CifnB = Variable("C if not b", boolean, position=(0.6,0.4))

p_A = Prob(A, [], [0.5,0.5])
p_B = Prob(B, [A, BifA, BifnA], [[[[1,0],[0,1]],[[1,0],[0,1]]], # A=0
                                  [[[1,0],[1,0]],[[0,1],[0,1]]]]) # A=1
p_C = Prob(C, [B, CifB, CifnB], [[[[1,0],[0,1]],[[1,0],[0,1]]], # B=0
                                  [[[1,0],[1,0]],[[0,1],[0,1]]]]) # B=1
p_Aprime = Prob(Aprime,[], [0.6,0.4])
p_Bprime = Prob(Bprime, [Aprime, BifA, BifnA], [[[[1,0],[0,1]],[[1,0],[0,1]]], # A=0
                                  [[[1,0],[1,0]],[[0,1],[0,1]]]]) # A=1
p_Cprime = Prob(Cprime, [Bprime, CifB, CifnB], [[[[1,0],[0,1]],[[1,0],[0,1]]], # B=0
                                  [[[1,0],[1,0]],[[0,1],[0,1]]]]) # B=1
p_bifa = Prob(BifA, [], [0.6,0.4]) # Does not actually depend on A!!!
p_bifna = Prob(BifnA, [], [0.6,0.4])
p_cifb = Prob(CifB, [], [0.9,0.1])
p_cifnb = Prob(CifnB, [], [0.2,0.8])

abcCounter = BeliefNetwork("ABC Counterfactual Example",
                        [A,B,C,Aprime,Bprime,Cprime,BifA, BifnA, CifB, CifnB],
                        [p_A,p_B,p_C,p_Aprime,p_Bprime, p_Cprime, p_bifa, p_bifna, p_cifb, p_cifnb])

abcq = ProbRC(abcCounter)
# abcq.queryDo(Cprime, obs = {Aprime:False, A:True})
# abcq.queryDo(Cprime, obs = {C:True, Aprime:False})
# abcq.queryDo(Cprime, obs = {A:True, C:True, Aprime:False})
# abcq.queryDo(Cprime, obs = {A:True, C:True, Aprime:False})
# abcq.queryDo(Cprime, obs = {A:False, C:True, Aprime:False})
# abcq.queryDo(CifB, obs = {C:True,Aprime:False})
# abcq.queryDo(CifnB, obs = {C:True,Aprime:False})

# abcq.show_post(obs = {})
# abcq.show_post(obs = {Aprime:False, A:True})
# abcq.show_post(obs = {A:True, C:True, Aprime:False})
# abcq.show_post(obs = {A:True, C:True, Aprime:True})
```

The following is the firing squad example of Pearl. See Figure 11.2.

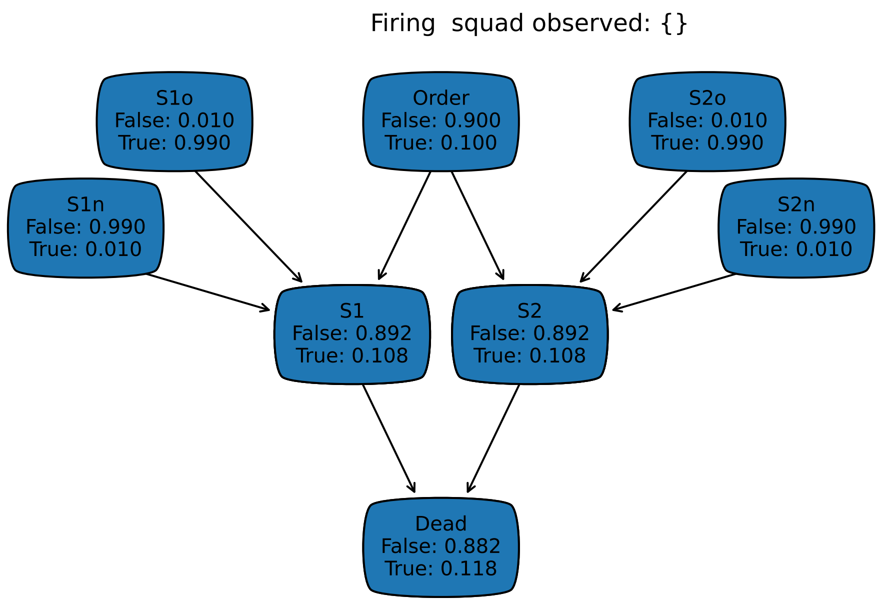


Figure 11.2: Firing squad belief network

probCounterfactual.py — (continued)

```
Order = Variable("Order", boolean, position=(0.4,0.8))
S1 = Variable("S1", boolean, position=(0.3,0.4))
S1o = Variable("S1o", boolean, position=(0.1,0.8))
S1n = Variable("S1n", boolean, position=(0.0,0.6))
S2 = Variable("S2", boolean, position=(0.5,0.4))
S2o = Variable("S2o", boolean, position=(0.7,0.8))
S2n = Variable("S2n", boolean, position=(0.8,0.6))
Dead = Variable("Dead", boolean, position=(0.4,0.0))

p_S1 = Prob(S1, [Order, S1o, S1n], [[[[1,0],[0,1]],[[1,0],[0,1]]], # Order=0
                                  [[[1,0],[1,0]],[[0,1],[0,1]]]]) # Order=1
p_S2 = Prob(S2, [Order, S2o, S2n], [[[[1,0],[0,1]],[[1,0],[0,1]]], # Order=0
                                  [[[1,0],[1,0]],[[0,1],[0,1]]]]) # Order=1
p_dead = Prob(Dead, [S1,S2], [[[1,0],[0,1]],[[0,1],[0,1]]])
p_order = Prob(Order, [], [0.9, 0.1])
p_s1o = Prob(S1o, [], [0.01, 0.99])
p_s1n = Prob(S1n, [], [0.99, 0.01])
p_s2o = Prob(S2o, [], [0.01, 0.99])
p_s2n = Prob(S2n, [], [0.99, 0.01])

firing_squad = BeliefNetwork("Firing squad",
```

```
                         [Order, S1, S1o, S1n, S2, S2o, S2n, Dead],
                         [p_order, p_dead, p_S1, p_s1o, p_s1n, p_S2, p_s2o,
                             p_s2n])
fsq = ProbRC(firing_squad)
# fsq.queryDo(Dead)
# fsq.queryDo(Order, obs={Dead:True})
# fsq.queryDo(Dead, obs={Order:True})
```

Chapter 12

Planning with Uncertainty

12.1 Decision Networks

The decision network code builds on the representation for belief networks of Chapter 9.

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 tabular factor – a list that enumerates the values – as in Section 9.3.3.

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)
```

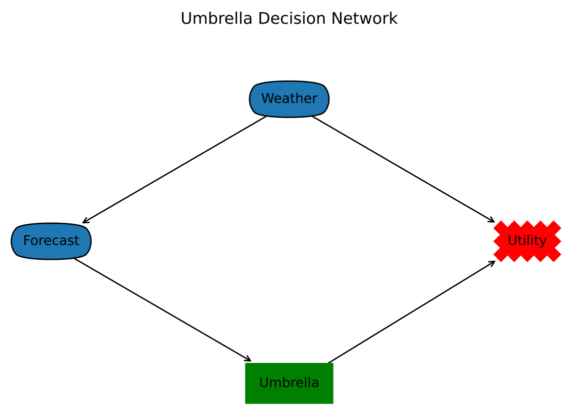


Figure 12.1: The umbrella decision network

```
        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
                                ha='center')
        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')
```

12.1.1 Example Decision Networks

Umbrella Decision Network

Here is a simple "umbrella" decision network. The output of `umbrella_dn.show()` is shown in Figure 12.1.

decnNetworks.py — (continued)

```
Weather = Variable("Weather", ["NoRain", "Rain"], position=(0.5,0.8))
Forecast = Variable("Forecast", ["Sunny", "Cloudy", "Rainy"],
    position=(0,0.4))
# Each variant uses one of the following:
```

```
Umbrella = DecisionVariable("Umbrella", ["Take", "Leave"], {Forecast},
    position=(0.5,0))

p_weather = Prob(Weather, [], [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))

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 12.2 (showing the result of `fire_dn.show()`) is represented as:

decnNetworks.py — (continued)

```
boolean = [False, True]
Alarm = Variable("Alarm", boolean, position=(0.25,0.633))
Fire = Variable("Fire", boolean, position=(0.5,0.9))
Leaving = Variable("Leaving", boolean, position=(0.25,0.366))
Report = Variable("Report", boolean, position=(0.25,0.1))
Smoke = Variable("Smoke", boolean, position=(0.75,0.633))
Tamper = Variable("Tamper", boolean, position=(0,0.9))

See_Sm = Variable("See_Sm", boolean, position=(0.75,0.366) )
Chk_Sm = DecisionVariable("Chk_Sm", boolean, {Report}, position=(0.5,
    0.366))
Call = DecisionVariable("Call", boolean,{See_Sm,Chk_Sm,Report},
    position=(0.75,0.1))

f_ta = Prob(Tamper,[],[0.98,0.02])
f_fi = Prob(Fire,[],[0.99,0.01])
f_sm = Prob(Smoke,[Fire],[[0.99,0.01],[0.1,0.9]])
f_al = Prob(Alarm,[Fire,Tamper],[[[0.9999, 0.0001], [0.15, 0.85]], [[0.01,
    0.99], [0.5, 0.5]]])
```

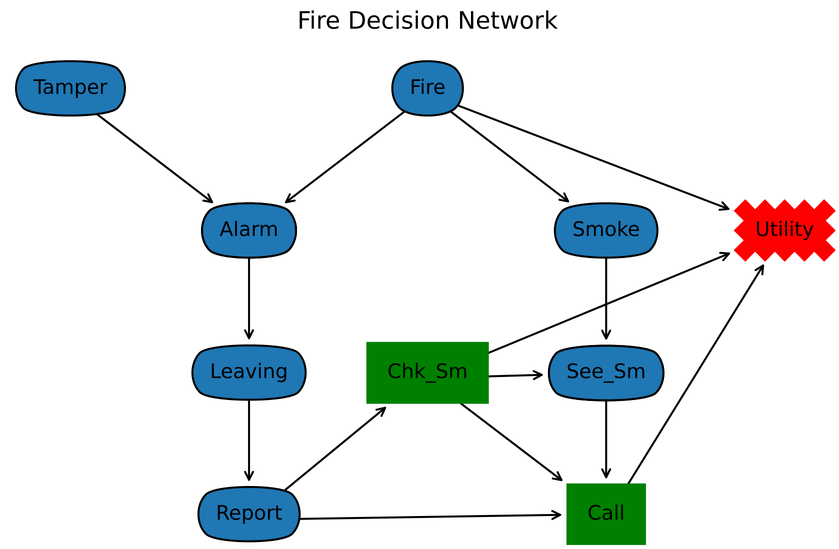


Figure 12.2: Fire Decision Network

```
f_lv = Prob(Leaving,[Alarm],[[0.999, 0.001], [0.12, 0.88]])
f_re = Prob(Report,[Leaving],[[0.99, 0.01], [0.25, 0.75]])
f_ss = Prob(See_Sm,[Chk_Sm,Smoke],[[[1,0],[1,0]],[[1,0],[0,1]]])

ut =
    UtilityTable([Chk_Sm,Fire,Call],[[[0,-200],[-5000,-200]],[[-20,-220],[-5020,-220]]],
    position=(1,0.633))

fire_dn = DecisionNetwork("Fire Decision Network",
                        {Tamper,Fire,Alarm,Leaving,Smoke,Call,See_Sm,Chk_Sm,Report},
                        {f_ta,f_fi,f_sm,f_al,f_lv,f_re,f_ss,ut})
```

Cheating Decision Network

The following is the representation of the cheating decision of Figure 12.3. 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))
```

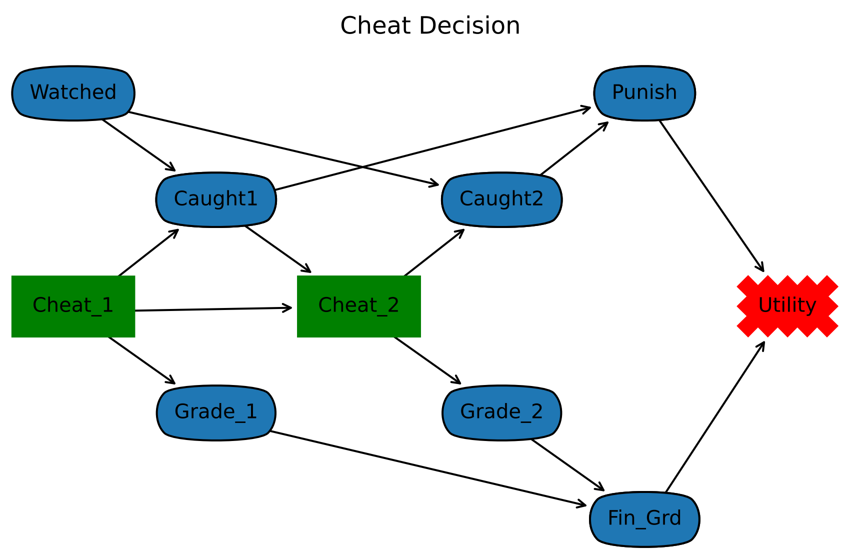


Figure 12.3: Cheating Decision Network

```
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, 0.5, 0.3]]])
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, 'D':0.0}])
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, 'D':0.0}])
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},
```

```
            '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))

cheating_dn = DecisionNetwork("Cheating Decision Network",
                         {Punish,Caught2,Watched,Fin_Grd,Grade_2,Grade_1,Cheat_2,Caught1,Cheat_1},
                         {p_wa, p_cc1, p_cc2, p_pun, p_gr1, p_gr2,p_fg,utc})
```

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 12.4.

decnNetworks.py — (continued)

```
S0 = Variable('S0', boolean, position=(0,0.5))
D0 = DecisionVariable('D0', boolean, {S0}, position=(1/7,0.1))
S1 = Variable('S1', boolean, position=(2/7,0.5))
D1 = DecisionVariable('D1', boolean, {S1}, position=(3/7,0.1))
S2 = Variable('S2', boolean, position=(4/7,0.5))
D2 = DecisionVariable('D2', boolean, {S2}, position=(5/7,0.1))
S3 = Variable('S3', boolean, position=(6/7,0.5))

p_s0 = Prob(S0, [], [0.5,0.5])
tr = [[[0.1, 0.9], [0.9, 0.1]], [[0.2, 0.8], [0.8, 0.2]]] # 0 is flip, 1 is keep value
p_s1 = Prob(S1, [D0,S0], tr)
p_s2 = Prob(S2, [D1,S1], tr)
p_s3 = Prob(S3, [D2,S2], tr)
```

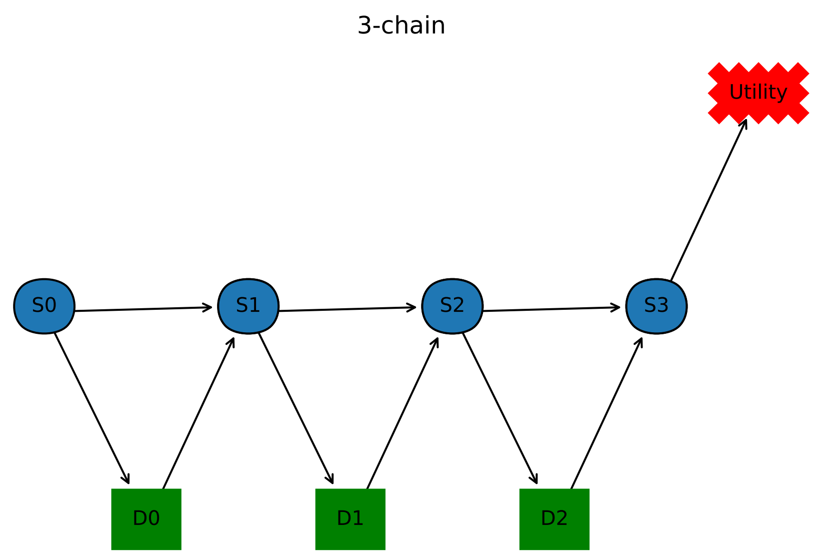


Figure 12.4: A decision network that is a chain of 3 decisions

```

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
```

12.1.2 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 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 {context}"
                maxres = -math.inf
                for val in var.domain:
                    self.display(3,"In rc0, branching on",var,"=",val)
                    newres = self.rc0({var:val}|context, factors, split_order[1:])
                    if newres > maxres:
                        maxres = newres
                        theval = val
                self.opt_policy[frozenset(context.items())] = (var,theval)
```

```
                return maxres
            else:
                total = 0
                for val in var.domain:
                    total += self.rc0({var:val}|context, factors, split_order[1:])
                self.display(3, "rc0 branching on", var,"returning", total)
                return total
```

We can combine the optimization for decision networks above, with the improvements of recursive conditioning used for graphical models (Section 9.7, page 201).

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({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({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(cheating_dn)
#rc_cheat.optimize()
#rc_cheat.opt_policy

#rc_ch3 = RC_DN(ch3)
#rc_ch3.optimize()
#rc_ch3.opt_policy
```

12.1.3 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.
    """
```

```python

    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)

```python
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)

```python
# 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(cheating_dn).optimize(); print(v)
# for df in p: print(df,"\n") # print decision functions
```

12.2 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

```
import random
import matplotlib.pyplot as plt
from matplotlib.widgets import Button, CheckButtons, TextBox
from display import Displayable
from utilities import argmaxd

class MDP(Displayable):
    """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
import matplotlib.pyplot as plt

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 MDPtiny(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_dim-1):0.25}
        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
```

12.2.1 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
        """
        self.display(3,"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(discount=0.9)
# gr.show()
# q,v,pi = gr.vi(100)
# q[(7,2)]
```

12.2.2 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([0.65,0.05,0.1,0.075]), "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.font_box = TextBox(plt.axes([0.1,0.05,0.05,0.075]),"Font:", textalignment="center")
        self.font_box.on_submit(self.set_font_size)
        self.font_box.set_val(str(plt.rcParams['font.size']))
        self.show_vals(None)
        plt.show()

    def set_font_size(self, s):
        plt.rcParams.update({'font.size': eval(s)})
        plt.draw()

    def show_vals(self,event):
        self.ax.cla()
        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)
```

```python

    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 12.5 shows the user interface for the tiny domain, which can be obtained using
`MDPtiny(discount=0.9).show()`
resizing it, checking "show q-values" and "show policy", and clicking "step" a few times.

> To run the demo in class do:
> `% python -i mdpExamples.py`
> `MDPtiny(discount=0.9).show()`

Figure 12.6 shows the user interface for the grid domain, which can be obtained using
`grid(discount=0.9).show()`
resizing it, checking "show q-values" and "show policy", and clicking "step" a few times.

Exercise 12.1 Computing q before v may seem like a waste of space because we don't need to store q in order to compute value function or the policy. Change the algorithm so that it loops through the states and actions once per iteration, and only stores the value function and the policy. Note that to get the same results as before, you would need to make sure that you use the previous value of v in the computation not the current value of v. Does using the current value of v hurt the algorithm or make it better (in approaching the actual value function)?

12.2.3 Asynchronous Value Iteration

This implements asynchronous value iteration, storing Q.

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)

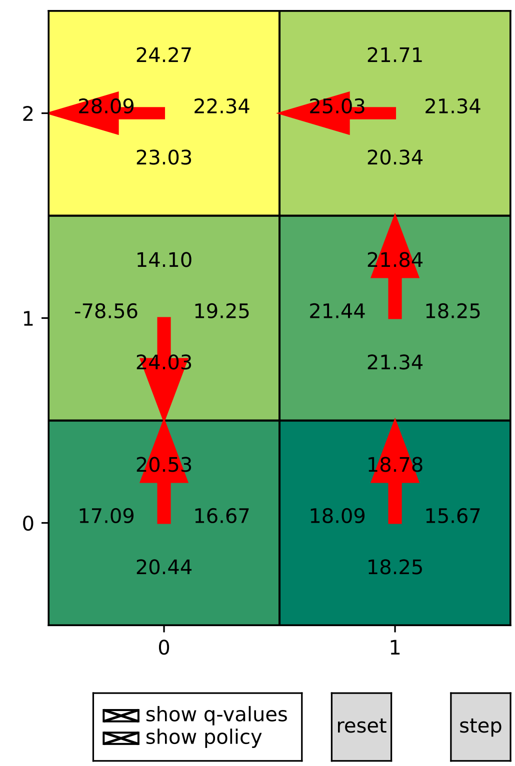


Figure 12.5: Interface for tiny example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The leftmost number is the for the left action; the rightmost number is for the right action; the upper most is for the *upR* (up-risky) action and the lowest number is for the *upC* action. The arrow points to the action(s) with the maximum Q-value. Use `MDPtiny().show()` after loading `mdpExamples.py`

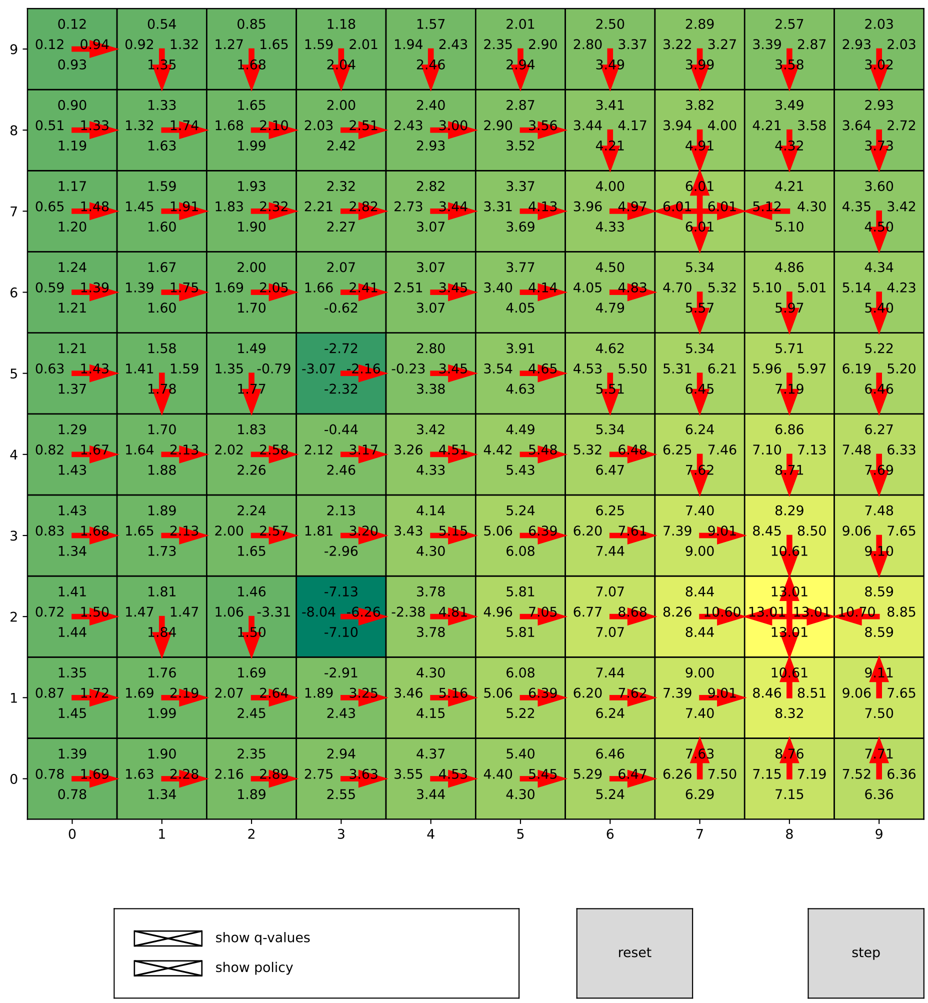


Figure 12.6: Interface for grid example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The leftmost number is the for the left action; the rightmost number is for the right action; the upper most is for the up action and the lowest number is for the down action. The arrow points to the action(s) with the maximum Q-value. From `grid(discount=0.9).show()`

```
    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(discount=0.9)
# q = gr.avi(100000)
# q[(7,2)]
```

Exercise 12.2 Implement value iteration that stores the V-values rather than the Q-values. Does it work better than storing Q? (What might better mean?)

Exercise 12.3 In asynchronous value iteration, try a number of different ways to choose the states and actions to update (e.g., sweeping through the state-action pairs, choosing them at random). Note that the best way may be to determine which states have had their Q-values 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 13

Reinforcement Learning

13.1 Representing Agents and Environments

The reinforcement learning agents and environments are instances of the general agent architecture of Section 2.1, where the percepts are reward–state pairs. The *state* is the world state; this is the fully observable assumption. In particular:

- An agent implements the method `select_action` that takes the reward and environment state and returns the next action (and updates the state of the agent).
- An environment implements the method `do` that takes the action and returns a pair of the reward and the resulting environment state.

These are chained together to simulate the system.

The only difference between the architecture of Section 2.1 is that the simulation starts by calling the agent method `initial_action(state)`, which is generally to save the state and return a random action.

The environments have names for the roles of agents participating. In this chapter, where we assume a single agent, this is used as the name of the environment.

rlProblem.py — Representations for Reinforcement Learning

```python
import random
import math
from display import Displayable
from agents import Agent, Environment
from utilities import pick_from_dist, argmaxe, argmaxd, flip

class RL_env(Environment):
```

```
    def __init__(self, name, actions, state):
        """creates an environment given name, list of actions, and initial
            state"""
        self.name = name       # the role for an agent
        self.actions = actions # list of all actions
        self.state = state     # initial state

    # must implement do(action)->(reward,state)
```

rlProblem.py — (continued)

```

class RL_agent(Agent):
    """An RL_Agent
    has percepts (s, r) for some state s and real reward r
    """
    def __init__(self, actions):
       self.actions = actions

    def initial_action(self, env_state):
        """return the initial action, and remember the state and action
        Act randomly initially
        Could be overridden to initialize data structures (as the agent now
            knows about one state)
        """
        self.state = env_state
        self.action = random.choice(self.actions)
        return self.action

    def select_action(self, reward, state):
        """
        Select the action given the reward and next state
        Remember the action in self.action
        This implements "Act randomly" and should be overridden!
        """
        self.action = random.choice(self.actions)
        return self.action
```

This is similar to `Simulate` of Section 2.1, except it is initialized by `agent.initial_action(state)`.

rlProblem.py — (continued)

```
import matplotlib.pyplot as plt

class Simulate(Displayable):
    """simulate the interaction between the agent and the environment
    for n time steps.
    Returns a pair of the agent state and the environment state.
    """
    def __init__(self, agent, environment):
        self.agent = agent
        self.env = environment
        self.action = agent.initial_action(self.env.state)
```

```
        self.reward_history = [] # for plotting

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

The following plots the sum of rewards as a function of the step in a simulation.

rlProblem.py — (continued)

```
    def plot(self, label=None, step_size=None, xscale='linear'):
        """
        plots the rewards history in the simulation
        label is the label for the plot
        step_size is the number of steps between each point plotted
        xscale is 'log' or 'linear'

        returns sum of rewards
        """
        if step_size is None: #for long simulations (> 999), only plot some points
            step_size = max(1,len(self.reward_history)//500)
        if label is None:
            label = self.agent.method
        plt.ion()
        plt.xscale(xscale)
        plt.xlabel("step")
        plt.ylabel("Sum of rewards")
        sum_history, sum_rewards = acc_rews(self.reward_history, step_size)
        plt.plot(range(0,len(self.reward_history),step_size), sum_history, label=label)
        plt.legend()
        plt.draw()
        return sum_rewards

def acc_rews(rews,step_size):
    """returns the rolling sum of the values, sampled each step_size, and the sum
    """
    acc = []
    sumr = 0; i=0
    for e in rews:
       sumr += e
       i += 1
       if (i%step_size == 0): acc.append(sumr)
    return acc, sumr
```

Here is the definition of the simple 2-state, 2-action decision about whether to party or relax (Example 12.29 in Poole and Mackworth [2023]).

rlExamples.py — Some example reinforcement learning environments

```
from rlProblem import RL_env
class Healthy_env(RL_env):
    def __init__(self):
        RL_env.__init__(self, "Party Decision", ["party", "relax"],
            "healthy")

    def do(self, action):
        """updates the state based on the agent doing action.
        returns reward,state
        """
        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 reward, self.state
```

13.1.1 Simulating an environment from an MDP

Given the definition for an MDP (page 264), *Env_from_MDP* takes in an MDP and simulates the environment with those dynamics.

Note that the representation of an MDP does not contain enough information to simulate a system, because does not specify the distribution over initial states, and it loses any dependency between the rewards and the resulting state (e.g., hitting the wall and having a negative reward may be correlated). The following code assumes 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 = random.choice(mdp.states)
        RL_env.__init__(self, "From MDP", mdp.actions, initial_state)
        self.mdp = mdp

    def do(self, action):
```

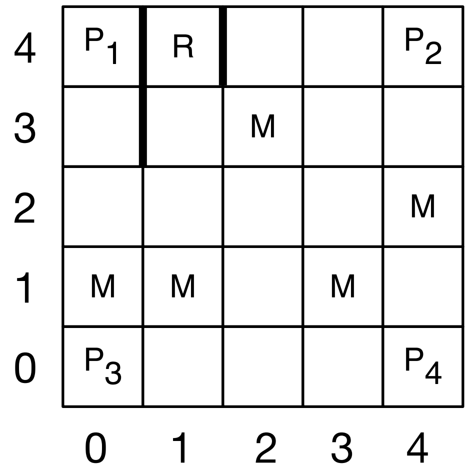


Figure 13.1: Monster game

```
        """updates the state based on the agent doing action.
        returns state,reward
        """
        reward = self.mdp.R(self.state,action)
        self.state = pick_from_dist(self.mdp.P(self.state,action))
        return reward,self.state
```

13.1.2 Monster Game

This is for the game depicted in Figure 13.1 (Example 13.2 of Poole and Mackworth [2023]).

rlExamples.py — (continued)

```
import random
from utilities import flip
from rlProblem import RL_env

class Monster_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.accumulated_rewards = 0 # sum of rewards received
        self.min_accumulated_rewards = 0
        self.min_step = 0
        self.zero_crossing = 0
        RL_env.__init__(self, "Monster Game", self.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 reward,state
        """
        assert action in self.actions, f"Monster game, unknown action:
            {action}"
        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(f"unknown_direction: {actual_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.accumulated_rewards += reward
        if self.accumulated_rewards < self.min_accumulated_rewards:
            self.min_accumulated_rewards = self.accumulated_rewards
            self.min_step = self.number_steps
        if self.accumulated_rewards>0 and reward>self.accumulated_rewards:
            self.zero_crossing = self.number_steps
        self.display(2,"",self.number_steps,self.accumulated_rewards,
                      self.accumulated_rewards/self.number_steps,sep="\t")

        return reward, (self.x, self.y, self.damaged, self.prize)
```

13.2 Q Learning

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

rlQLearner.py — Q Learning

```
import random
import math
```

```python
from display import Displayable
from utilities import argmaxe, argmaxd, flip
from rlProblem import RL_agent, epsilon_greedy, ucb

class Q_learner(RL_agent):
    """A Q-learning agent has
    belief-state consisting of
        state is the previous state (initialized by RL_agent
        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

    """
```

rlQLearner.py — (continued)

```python
    def __init__(self, role, actions, discount,
                 exploration_strategy=epsilon_greedy, es_kwargs={},
                 alpha_fun=lambda _:0.2,
                 Qinit=0, method="Q_learner"):
        """
        role is the role of the agent (e.g., in a game)
        actions is the set of actions the agent can do
        discount is the discount factor
        exploration_strategy is the exploration function, default
            "epsilon_greedy"
        es_kwargs is extra arguments of exploration_strategy
        alpha_fun is a function that computes alpha from the number of
            visits
        Qinit is the initial q-value
        method gives the method used to implement the role (for plotting)
        """
        RL_agent.__init__(self, actions)
        self.role = role
        self.discount = discount
        self.exploration_strategy = exploration_strategy
        self.es_kwargs = es_kwargs
        self.alpha_fun = alpha_fun
        self.Qinit = Qinit
        self.method = method
        self.acc_rewards = 0
        self.Q = {}
        self.visits = {}
```

The initial action is a random action. It remembers the state, and initializes the data structures.

rlQLearner.py — (continued)

```python
    def initial_action(self, state):
        """ Returns the initial action; selected at random
        Initialize Data Structures
```

```
        """
        self.Q[state] = {act:self.Qinit for act in self.actions}
        self.action = random.choice(self.actions)
        self.visits[state] = {act:0 for act in self.actions}
        self.state = state
        self.display(2, f"Initial State: {state} Action {self.action}")
        self.display(2,"s\ta\tr\ts'\tQ")
        return self.action

    def select_action(self, reward, next_state):
        """give reward and next state, select next action to be carried
            out"""
        if next_state not in self.visits: # next state not seen before
            self.Q[next_state] = {act:self.Qinit for act in self.actions}
            self.visits[next_state] = {act:0 for act in self.actions}
        self.visits[self.state][self.action] +=1
        alpha = self.alpha_fun(self.visits[self.state][self.action])
        self.Q[self.state][self.action] += alpha*(
                            reward
                            + self.discount * max(self.Q[next_state].values())
                            - self.Q[self.state][self.action])
        self.display(2,self.state, self.action, reward, next_state,
                    self.Q[self.state][self.action], sep='\t')
        self.state = next_state
        self.action = self.exploration_strategy(self.Q[next_state],
                                    self.visits[next_state],**self.es_kwargs)
        self.display(3,f"Agent {self.role} doing {self.action} in state
            {self.state}")
        return self.action
```

Exercise 13.1 Implement SARSA. Hint: it does not do a *max* in *do*. Instead it needs to choose *next_act* before it does the update.

13.2.1 Exploration Strategies

Two explorations strategies are defined: epsilon-greedy and UCB.

In general an exploration strategy takes two arguments

- *Qs* is a $\{action : q\_value\}$ dictionary for the current state
- *Vs* is a $\{action : visits\}$ dictionary for the current state; where *visits* is the number of times that the action has been carried out in the current state.

rlProblem.py — (continued)

```
def epsilon_greedy(Qs, Vs={}, epsilon=0.1):
        """select action given epsilon greedy
        Qs is the {action:Q-value} dictionary for current state
        Vs is ignored
        """
```

```
        if flip(epsilon):
            return random.choice(list(Qs.keys())) # act randomly
        else:
            return argmaxd(Qs)

def ucb(Qs, Vs, c=1.4):
        """select action given upper-confidence bound
        Qs is the {action:Q-value} dictionary for current state
        Vs is the {action:visits} dictionary for current state

        0.01 is to prevent divide-by zero (could just be infinity)
        """
        Ns = sum(Vs.values())
        ucb1 = {a:Qs[a]+c*math.sqrt(Ns/(0.01+Vs[a]))
                    for a in Qs.keys()}
        action = argmaxd(ucb1)
        return action
```

Exercise 13.2 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.

13.2.2 Testing Q-learning

The first tests are for the 2-action 2-state decision about whether to relax or party (Example 12.29 of Poole and Mackworth [2023].

rlQLearner.py — (continued)

```
####### TEST CASES ########
from rlProblem import Simulate,epsilon_greedy, ucb
from rlExamples import Healthy_env, Monster_game_env
from rlQLearner import Q_learner

env = Healthy_env()
# Some RL agents with different parameters:
ag = Q_learner(env.name, env.actions, 0.7)
ag_ucb = Q_learner(env.name, env.actions, 0.7, exploration_strategy = ucb,
    es_kwargs={'c':0.1}, method="ucb")
ag_opt = Q_learner(env.name, env.actions, 0.7, Qinit=100,
    method="optimistic" )
ag_exp_m = Q_learner(env.name, env.actions, 0.7,
    es_kwargs={'epsilon':0.5}, method="more explore")
ag_greedy = Q_learner(env.name, env.actions, 0.1, Qinit=100, method="disc
    0.1")

sim_ag = Simulate(ag,env)

# sim_ag.go(100)
# ag.Q   # get the learned Q-values
# sim_ag.plot()
```

```
# Simulate(ag_ucb,env).go(100).plot()
# Simulate(ag_opt,env).go(100).plot()
# Simulate(ag_exp_m,env).go(100).plot()
# Simulate(ag_greedy,env).go(100).plot()


from mdpExamples import MDPtiny
from rlProblem import Env_from_MDP
envt = Env_from_MDP(MDPtiny())
agt = Q_learner(envt.name, envt.actions, 0.8)
#Simulate(agt, envt).go(1000).plot()

mon_env = Monster_game_env()
mag1 = Q_learner(mon_env.name, mon_env.actions,0.9)
#Simulate(mag1,mon_env).go(100000).plot()
mag_ucb = Q_learner(mon_env.name, mon_env.actions,0.9,exploration_strategy
    = ucb,es_kwargs={'c':0.1},method="UCB(0.1)")
#Simulate(mag_ucb,mon_env).go(100000).plot()

mag2 = Q_learner(mon_env.name, mon_env.actions,
    0.9,es_kwargs={'epsilon':0.2},alpha_fun=lambda
    k:1/k,method="alpha=1/k")
#Simulate(mag2,mon_env).go(100000).plot()
mag3 = Q_learner(mon_env.name, mon_env.actions, 0.9,alpha_fun=lambda
    k:10/(9+k),method="alpha=10/(9+k)")
#Simulate(mag3,mon_env).go(100000).plot()
```

13.3 Q-leaning with Experience Replay

A bounded buffer remembers values up to size `buffer_size`. Once it is full, all old experiences have the same chance of being in the buffer.

rlQExperienceReplay.py — Q-Learner with Experience Replay

```
from rlQLearner import Q_learner
from utilities import flip
import random

class BoundedBuffer(object):
    def __init__(self, buffer_size=1000):
        self.buffer_size = buffer_size
        self.buffer = [0]*buffer_size
        self.number_added = 0

    def add(self,experience):
        if self.number_added < self.buffer_size:
            self.buffer[self.number_added] = experience
        else:
            if flip(self.buffer_size/self.number_added):
                position = random.randrange(self.buffer_size)
```

```
            self.buffer[position] = experience
        self.number_added += 1

    def get(self):
        return self.buffer[random.randrange(min(self.number_added,
            self.buffer_size))]
```

A Q_ER_Learner does Q-leaning with experience replay. It only uses action replay after burn_in number of steps.

rlQExperienceReplay.py — (continued)

```
class Q_ER_learner(Q_learner):
    def __init__(self, role, actions, discount,
                 max_buffer_size=10000,
                 num_updates_per_action=5, burn_in=1000,
                method="Q_ER_learner", **q_kwargs):
        """Q-learner with experience replay
        role is the role of the agent (e.g., in a game)
        actions is the set of actions the agent can do
        discount is the discount factor
        max_buffer_size is the maximum number of past experiences that is
            remembered
        burn_in is the number of steps before using old experiences
        num_updates_per_action is the number of q-updates for past
            experiences per action
        q_kwargs are any extra parameters for Q_learner
        """
        Q_learner.__init__(self, role, actions, discount, method=method,
            **q_kwargs)
        self.experience_buffer = BoundedBuffer(max_buffer_size)
        self.num_updates_per_action = num_updates_per_action
        self.burn_in = burn_in

    def select_action(self, reward, next_state):
        """give reward and new state, select next action to be carried
            out"""
        self.experience_buffer.add((self.state,self.action,reward,next_state))
            #remember experience
        if next_state not in self.Q: # Q and visits are defined on the same
            states
            self.Q[next_state] = {act:self.Qinit for act in self.actions}
            self.visits[next_state] = {act:0 for act in self.actions}
        self.visits[self.state][self.action] +=1
        alpha = self.alpha_fun(self.visits[self.state][self.action])
        self.Q[self.state][self.action] += alpha*(
                            reward
                            + self.discount * max(self.Q[next_state].values())
                            - self.Q[self.state][self.action])
        self.display(2,self.state, self.action, reward, next_state,
                    self.Q[self.state][self.action], sep='\t')
        self.state = next_state
```

```
        # do some updates from experience buffer
        if self.experience_buffer.number_added > self.burn_in:
            for i in range(self.num_updates_per_action):
                (s,a,r,ns) = self.experience_buffer.get()
                self.visits[s][a] +=1 # is this correct?
                alpha = self.alpha_fun(self.visits[s][a])
                self.Q[s][a] += alpha * (r +
                                    self.discount* max(self.Q[ns][na]
                                            for na in self.actions)
                                    -self.Q[s][a] )
        ### CHOOSE NEXT ACTION ###
        self.action = self.exploration_strategy(self.Q[next_state],
                                    self.visits[next_state],**self.es_kwargs)
        self.display(3,f"Agent {self.role} doing {self.action} in state
            {self.state}")
        return self.action
```

rlQExperienceReplay.py — (continued)

```
from rlProblem import Simulate
from rlExamples import Monster_game_env
from rlQLearner import mag1, mag2, mag3

mon_env = Monster_game_env()
mag1ar = Q_ER_learner(mon_env.name, mon_env.actions,0.9,method="Q_ER")
# Simulate(mag1ar,mon_env).go(100000).plot()

mag3ar = Q_ER_learner(mon_env.name, mon_env.actions, 0.9, alpha_fun=lambda
    k:10/(9+k),method="Q_ER alpha=10/(9+k)")
# Simulate(mag3ar,mon_env).go(100000).plot()
```

13.4 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 state s and action a returns the Q-value, the estimate of the future (discounted) value of being in state s and doing action a.

- $R[s][a]$ is dictionary that, given a (s, a) state s and action a is the average reward received from doing a in state s.

- $T[s][a][s']$ is dictionary that, given states s and s' and action a returns the number of times a was done in state s and the result was state s'. Note that s' is only a key if it has been the result of doing a in s; there are no 0 counts recorded.

- $visits[s][a]$ is dictionary that, given state s and action a returns the number of times action a was carried out in state s. This is the C of Figure 13.6 of Poole and Mackworth [2023].

  Note that $visits[s][a] = \sum_{s'} T[s][a][s']$ but is stored separately to keep the code more readable.

The main difference to Figure 13.6 of Poole and Mackworth [2023] is the code below does a fixed number of asynchronous value iteration updates per step.

rlModelLearner.py — Model-based Reinforcement Learner

```python
import random
from rlProblem import RL_agent, Simulate, epsilon_greedy, ucb
from display import Displayable
from utilities import argmaxe, flip

class Model_based_reinforcement_learner(RL_agent):
    """A Model-based reinforcement learner
    """

    def __init__(self, role, actions, discount,
                     exploration_strategy=epsilon_greedy, es_kwargs={},
                     Qinit=0,
                   updates_per_step=10, method="MBR_learner"):
        """role is the role of the agent (e.g., in a game)
        actions is the list of actions the agent can do
        discount is the discount factor
        explore is the proportion of time the agent will explore
        Qinit is the initial value of the Q's
        updates_per_step is the number of AVI updates per action
        label is the label for plotting
        """
        RL_agent.__init__(self, actions)
        self.role = role
        self.actions = actions
        self.discount = discount
        self.exploration_strategy = exploration_strategy
        self.es_kwargs = es_kwargs
        self.Qinit = Qinit
        self.updates_per_step = updates_per_step
        self.method = method
```

rlModelLearner.py — (continued)

```python
    def initial_action(self, state):
        """ Returns the initial action; selected at random
```

```
        Initialize Data Structures

        """
        self.state = state
        self.T = {self.state: {a: {} for a in self.actions}}
        self.visits = {self.state: {a: 0 for a in self.actions}}
        self.Q = {self.state: {a: self.Qinit for a in self.actions}}
        self.R = {self.state: {a: 0 for a in self.actions}}
        self.states_list = [self.state] # list of states encountered
        self.action = random.choice(self.actions)
        self.display(2, f"Initial State: {state} Action {self.action}")
        self.display(2,"s\ta\tr\ts'\tQ")
        return self.action
```

rlModelLearner.py — (continued)

```
    def select_action(self, reward, next_state):
        """do num_steps of interaction with the environment
        for each action, do updates_per_step iterations of asynchronous
            value iteration
        """
        if next_state not in self.visits: # has not been encountered before
            self.states_list.append(next_state)
            self.visits[next_state] = {a:0 for a in self.actions}
            self.T[next_state] = {a:{} for a in self.actions}
            self.Q[next_state] = {a:self.Qinit for a in self.actions}
            self.R[next_state] = {a:0 for a in self.actions}
        if next_state in self.T[self.state][self.action]:
            self.T[self.state][self.action][next_state] += 1
        else:
            self.T[self.state][self.action][next_state] = 1
        self.visits[self.state][self.action] += 1
        self.R[self.state][self.action] +=
            (reward-self.R[self.state][self.action])/self.visits[self.state][self.action]
        st,act = self.state,self.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][nst]/self.visits[st][act]*
                    max(self.Q[nst][nact] for nact in self.actions)
                    for nst in self.T[st][act].keys()))
            st = random.choice(self.states_list)
            act = random.choice(self.actions)
        self.state = next_state
        self.action = self.exploration_strategy(self.Q[next_state],
                              self.visits[next_state],**self.es_kwargs)
        return self.action
```

rlModelLearner.py — (continued)

```
from rlExamples import Monster_game_env
mon_env = Monster_game_env()
```

```
mbl1 = Model_based_reinforcement_learner(mon_env.name, mon_env.actions,
    0.9, updates_per_step=10,method="model-based(10)")
# Simulate(mbl1,mon_env).go(100000).plot()
mbl2 = Model_based_reinforcement_learner(mon_env.name, mon_env.actions,
    0.9, updates_per_step=1, method="model-based(1)")
# Simulate(mbl2,mon_env).go(100000).plot()
```

Exercise 13.3 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 13.4 It is possible to implement the model-based reinforcement learner by replacing *Q*, *R*, *T*, *visits*, *res_states* with a single dictionary that, given a state and action returns a tuple corresponding to these data structures. Does this make the algorithm easier to understand? Does this make the algorithm more efficient?

Exercise 13.5 If the states and the actions were mapped into integers, the dictionaries could be implemented perhaps more efficiently as arrays. How does the code need to change?. Implement this for the monster game. Is it more efficient?

Exercise 13.6 In `random_choice` in the updates of `select_action`, all state-action pairs have the same chance of being chosen. Does selecting state-action pairs proportionally to the number of times visited work better than what is implemented? Provide evidence for your answer.

13.5 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.

13.5.1 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 monster game.

rlMonsterGameFeatures.py — Feature-based Reinforcement Learner

```
from rlExamples import Monster_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 Monster_game_env.actions, f"Monster game, unknown
        action: {action}"
    (x,y,d,p) = state
    # f1: would go to a monster
    f1 = monster_ahead(x,y,action)
    # f2: would crash into wall
    f2 = wall_ahead(x,y,action)
    # f3: action is towards a prize
    f3 = towards_prize(x,y,action,p)
    # f4: damaged and action is toward repair station
    f4 = towards_repair(x,y,action) if d else 0
    # f5: damaged and towards monster
    f5 = 1 if d and f1 else 0
    # f6: damaged
    f6 = 1 if d else 0
    # f7: not damaged
    f7 = 1-f6
    # f8: damaged and prize ahead
    f8 = 1 if d and f3 else 0
    # f9: not damaged and prize ahead
    f9 = 1 if not d and f3 else 0
    features = [1,f1,f2,f3,f4,f5,f6,f7,f8,f9]
    # the next 20 features are for 5 prize locations
    # and 4 distances from outside in all directions
    for pr in Monster_game_env.prize_locs+[None]:
        if p==pr:
            features += [x, 4-x, y, 4-y]
        else:
            features += [0, 0, 0, 0]
    # fp04 feature for y when prize is at 0,4
    # this knows about the wall to the right of the prize
    if p==(0,4):
        if x==0:
            fp04 = y
        elif y<3:
            fp04 = y
        else:
            fp04 = 4-y
    else:
        fp04 = 0
    features.append(fp04)
    return features

def monster_ahead(x,y,action):
    """returns 1 if the location expected to get to by doing
    action from (x,y) can contain a monster.
    """
    if action == "right" and (x+1,y) in Monster_game_env.monster_locs:
        return 1
    elif action == "left" and (x-1,y) in Monster_game_env.monster_locs:
```

```python
        return 1
    elif action == "up" and (x,y+1) in Monster_game_env.monster_locs:
        return 1
    elif action == "down" and (x,y-1) in Monster_game_env.monster_locs:
        return 1
    else:
        return 0

def wall_ahead(x,y,action):
    """returns 1 if there is a wall in the direction of action from (x,y).
    This is complicated by the internal walls.
    """
    if action == "right" and (x==Monster_game_env.xdim-1 or (x,y) in
         Monster_game_env.vwalls):
        return 1
    elif action == "left" and (x==0 or (x-1,y) in Monster_game_env.vwalls):
        return 1
    elif action == "up" and y==Monster_game_env.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
```

The following uses a simpler set of features. In particular, it only considers whether the action will most likely result in a monster position or a wall, and whether the action moves towards the current prize.

rlMonsterGameFeatures.py — (continued)

```
def simp_features(state,action):
    """returns a list of feature values for the state-action pair
    """
    assert action in Monster_game_env.actions
    (x,y,d,p) = state
    # f1: would go to a monster
    f1 = monster_ahead(x,y,action)
    # f2: would crash into wall
    f2 = wall_ahead(x,y,action)
    # f3: action is towards a prize
    f3 = towards_prize(x,y,action,p)
    return [1,f1,f2,f3]
```

13.5.2 Feature-based RL learner

This learns a linear function approximation of the Q-values. It requires the function *get_features* that given a state and an action returns a list of values for all of the features. Each environment requires this function to be provided.

rlFeatures.py — Feature-based Reinforcement Learner

```
import random
from rlProblem import RL_agent, epsilon_greedy, ucb
from display import Displayable
from utilities import argmaxe, flip

class SARSA_LFA_learner(RL_agent):
    """A SARSA with linear function approximation (LFA) learning agent has
    """
    def __init__(self, role, actions, discount, get_features,
```

```
                 exploration_strategy=epsilon_greedy, es_kwargs={},
                 step_size=0.01, winit=0, method="SARSA_LFA"):
        """role is the role of the agent (e.g., in a game)
        actions is the set of actions the agent can do
        discount is the discount factor
        get_features is a function get_features(state,action) -> list of
            feature values
        exploration_strategy is the exploration function, default
            "epsilon_greedy"
        es_kwargs is extra keyword arguments of the exploration_strategy
        step_size is gradient descent step size
        winit is the initial value of the weights
        method gives the method used to implement the role (for plotting)
        """
        RL_agent.__init__(self, actions)
        self.role = role
        self.discount = discount
        self.exploration_strategy = exploration_strategy
        self.es_kwargs = es_kwargs
        self.get_features = get_features
        self.step_size = step_size
        self.winit = winit
        self.method = method
```

The initial action is a random action. It remembers the state, and initializes the data structures.

rlFeatures.py — (continued)

```
    def initial_action(self, state):
        """ Returns the initial action; selected at random
        Initialize Data Structures
        """
        self.action = random.choice(self.actions)
        self.features = self.get_features(state, self.action)
        self.weights = [self.winit for f in self.features]
        self.state = state
        self.display(2, f"Initial State: {state} Action {self.action}")
        self.display(2,"s\ta\tr\ts'\tQ")
        return self.action
```

do takes in the number of steps.

rlFeatures.py — (continued)

```

    def Q(self, state,action):
        """returns Q-value of the state and action for current weights
        """
        return dot_product(self.weights, self.get_features(state,action))

    def select_action(self, reward, next_state):
        """do num_steps of interaction with the environment"""
        feature_values = self.get_features(self.state,self.action)
```

```
        oldQ = self.Q(self.state,self.action)
        next_action = self.exploration_strategy({a:self.Q(next_state,a)
                                                 for a in self.actions}, {})
        nextQ = self.Q(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,
                     self.Q(self.state,self.action), delta, sep='\t')
        self.state = next_state
        self.action = next_action
        return self.action

    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 rlProblem import Simulate
from rlExamples import Monster_game_env # monster game environment
import rlMonsterGameFeatures

mon_env = Monster_game_env()
fa1 = SARSA_LFA_learner(mon_env.name, mon_env.actions, 0.9,
    rlMonsterGameFeatures.get_features)
# Simulate(fa1,mon_env).go(100000).plot()
fas1 = SARSA_LFA_learner(mon_env.name, mon_env.actions, 0.9,
    rlMonsterGameFeatures.simp_features, method="LFA (simp features)")
#Simulate(fas1,mon_env).go(100000).plot()
```

Exercise 13.7 How does the step-size affect performance? Try different step sizes (e.g., 0.1, 0.001, other sizes in between). Explain the behavior you observe. Which step size works best for this example. Explain what evidence you are basing your prediction on.

Exercise 13.8 Does having extra features always help? Does it sometime help? Does whether it helps depend on the step size? Give evidence for your claims.

Exercise 13.9 For each of the following first predict, then plot, then explain the behavior you observed:

(a) SARSA_LFA, Model-based learning (with 1 update per step) and Q-learning for 10,000 steps 20% exploring followed by 10,000 steps 100% exploiting

(b) SARSA_LFA, model-based learning and Q-learning for

   i) 100,000 steps 20% exploring followed by 100,000 steps 100% exploit

   ii) 10,000 steps 20% exploring followed by 190,000 steps 100% exploit

(c) Suppose your goal was to have the best accumulated reward after 200,000 steps. You are allowed to change the exploration rate at a fixed number of steps. For each of the methods, which is the best position to start exploiting more? Which method is better? What if you wanted to have the best reward after 10,000 or 1,000 steps?

Based on this evidence, explain when it is preferable to use SARSA_LFA, Model-based learner, or Q-learning.

Important: you need to run each algorithm more than once. Your explanation should include the variability as well as the typical behavior.

Exercise 13.10 In the call to `self.exploration_strategy`, what should the counts be? (The code above will fail for ucb, for example.) Think about the case where there are too many states. Suppose we are just learning for a neighborhood of a current state (e.g., a fixed number of steps away the from the current state); how could the algorithm be modifies to make sure it has at least explored the close neighborhood of the current state?

Chapter 14

Multiagent Systems

14.1 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.

14.1.1 Creating a two-player game

masProblem.py — A Multiagent Problem

```
from display import Displayable

class Node(Displayable):
    """A node in a search tree. It has a
    name a string
    isMax is True if it is a maximizing node, otherwise it is minimizing
        node
    children is the list of children
    value is what 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 14.1); 3 numbers that add to 15 correspond exactly to the winning positions

6	1	8
7	5	3
2	9	4

Figure 14.1: Magic Square

of tic-tac-toe played on the magic square.

Note that we do not remove symmetries. (What are the symmetries? How 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, together with two other elements of selected sum to
        15.
    """
    return any(last+a+b == 15
               for a in selected if a != last
               for b in selected if b != last and b != a)
```

14.1.2 Minimax and α-β Pruning

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

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)
```

14.2 Multiagent Learning

The next code of for multiple agents that learn when interacting with other agents. The main difference with the learners from the last chapter is that the games take actions from all agents and provide a separate reward to each agent.

The following agent maintains a stochastic policy; it learns a distribution over actions for each state.

masLearn.py — Simulations of agents learning

```
from display import Displayable
import utilities # argmaxall for (element,value) pairs
import matplotlib.pyplot as plt
import random
from rlProblem import RL_agent


class StochasticPIAgent(RL_agent):
    """This agent maintains the Q-function for each state.
    Chooses the best action using empirical distribution over actions
    """
    def __init__(self, role, actions, discount=0,
                 alpha_fun=lambda k:10/(9+k), Qinit=1, pi_init=1,
                     method="Stochastic Q_learner"):
        """
        role is the role of the agent (e.g., in a game)
        actions is the set of actions the agent can do.
        discount is the discount factor (0 is appropriate if there is a
            single state)
        alpha_fun is a function that computes alpha from the number of
            visits
```

```
        Qinit is the initial q-values
        pi_init gives the prior counts (Dirichlet prior) for the policy (must be >0)
        method gives the method used to implement the role
        """
        #self.max_display_level = 3
        RL_agent.__init__(self, actions)
        self.role = role
        self.discount = discount
        self.alpha_fun = alpha_fun
        self.Qinit = Qinit
        self.pi_init = pi_init
        self.method = method
        self.Q = {}
        self.pi = {}
        self.visits = {}

    def initial_action(self, state):
        """ Returns the initial action; selected at random
        Initialize Data Structures
        """
        self.Q[state] = {act:self.Qinit for act in self.actions}
        self.pi[state] = {act:self.pi_init for act in self.actions}
        self.action = random.choice(self.actions)
        self.visits[state] = {act:0 for act in self.actions}
        self.state = state
        self.display(2, f"Initial State: {state} Action {self.action}")
        self.display(2,"s\ta\tr\ts'\tQ")
        return self.action

    def select_action(self, reward, next_state):
        """give reward and next state, select next action to be carried out"""
        if next_state not in self.visits: # next state not seen before
            self.Q[next_state] = {act:self.Qinit for act in self.actions}
            self.pi[next_state] = {act:self.pi_init for act in
              self.actions}
            self.visits[next_state] = {act:0 for act in self.actions}
        self.visits[self.state][self.action] +=1
        alpha = self.alpha_fun(self.visits[self.state][self.action])
        self.Q[self.state][self.action] += alpha*(
                          reward
                          + self.discount * max(self.Q[next_state].values())
                          - self.Q[self.state][self.action])
        a_best = utilities.argmaxd(self.Q[self.state])
        self.pi[self.state][a_best] +=1
        self.display(2,self.state, self.action, reward, next_state,
                     self.Q[self.state][self.action], sep='\t')
        self.state = next_state
        self.action = select_from_dist(self.pi[next_state])
```

```
        self.display(3,f"Agent {self.role} doing {self.action} in state
            {self.state}")
        return self.action


def normalize(dist):
    """dict is a {value:number} dictionary, where the numbers are all
        non-negative
    returns dict where the numbers sum to one
    """
    tot = sum(dist.values())
    return {var:val/tot for (var,val) in dist.items()}

def select_from_dist(dist):
    rand = random.random()
    for (act,prob) in normalize(dist).items():
        rand -= prob
        if rand < 0:
            return act
```

The agent can be tested on the reinforcement learning benchmarks from the previous chapter:

masLearn.py — (continued)

```
#### Testing on RL benchmarks #####
from rlProblem import Simulate
from rlExamples import Healthy_env, Monster_game_env
mon_env = Monster_game_env()
magspi =StochasticPIAgent(mon_env.name, mon_env.actions,0.9)
#Simulate(magspi,mon_env).go(100000).plot()
```

The simulation for a game passes the joint action from all agents to the environment, which returns a tuple of rewards – one for each agent – and the next state.

masLearn.py — (continued)

```
class SimulateGame(Displayable):
    def __init__(self, game, agent_types):
        #self.max_display_level = 3
        self.game = game
        self.agents = [agent_types[i](game.players[i], game.actions[i], 0)
            for i in range(game.num_agents)] # list of agents
        self.action_dists = [{act:0 for act in game.actions[i]} for i in
            range(game.num_agents)]
        self.action_history = []
        self.state_history = []
        self.reward_history = []
        self.dist = {}
        self.dist_history = []
        self.actions = tuple(ag.initial_action(game.initial_state) for ag
            in self.agents)
```

```
        self.num_steps = 0

    def go(self, steps):
        for i in range(steps):
            self.num_steps += 1
            (self.rewards, state) = self.game.play(self.actions)
            self.display(3, f"In go rewards={self.rewards}, state={state}")
            self.reward_history.append(self.rewards)
            self.state_history.append(state)
            self.actions = tuple(agent.select_action(reward, state)
                                     for (agent,reward) in
                                         zip(self.agents,self.rewards))
            self.action_history.append(self.actions)
            for i in range(self.game.num_agents):
                 self.action_dists[i][self.actions[i]] += 1
            self.dist_history.append([{a:i for (a,i) in elt.items()} for
                elt in self.action_dists]) # deep copy
        #print("Scores:", ' '.join(f"{self.agents[i].role} average
            reward={ag.total_score/self.num_steps}" for ag in self.agents))
        print("Distributions:", '
            '.join(str({a:self.dist_history[-1][i][a]/sum(self.dist_history[-1][i].values())
            for a in self.game.actions[i]})
                                            for  i in
                                                range(self.game.num_agents)))
        #return self.reward_history, self.action_history

    def action_dist(self,which_actions=[1,1]):
        """ which actions is [a0,a1]
        returns the empirical distribution of actions for agents,
           where ai specifies the index of the actions for agent i
        remove this???
        """
        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)]
```

The plotting shows how the empirical distributions of the first two agents changes as the learning continues.

masLearn.py — (continued)

```
    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"Probability {self.game.players[0]}
            {self.agents[0].actions[x_action]}")
        plt.ylabel(f"Probability {self.game.players[1]}
            {self.agents[1].actions[y_action]}")
```

```
        plt.plot([self.dist_history[i][0][x_act]/sum(self.dist_history[i][0].values()) for i in range(len(self.dist_history))],
                [self.dist_history[i][1][y_act]/sum(self.dist_history[i][1].values()) for i in range(len(self.dist_history))])
        #plt.legend()
        plt.savefig('soccerplot.pdf')
        plt.show()
```

masLearn.py — (continued)

```

class ShoppingGame(Displayable):
    def __init__(self):
        self.num_agents = 2
        self.states = ['s']
        self.initial_state = 's'
        self.actions = [['shopping', 'football']]*2
        self.players = ['football-preferrer goes to', 'shopping-preferrer goes to']

    def play(self, actions):
        """Given (action1,action2) returns (resulting_state, (rewward1, reward2))
        """
        return ({('football', 'football'): (2, 1),
                ('football', 'shopping'): (0, 0),
                ('shopping', 'football'): (0, 0),
                ('shopping', 'shopping'): (1, 2)
                    }[actions], 's')


class SoccerGame(Displayable):
    def __init__(self):
        self.num_agents = 2
        self.states = ['s']
        self.initial_state = 's'
        self.initial_state = 's'
        self.actions = [['right', 'left']]*2
        self.players = ['goalkeeper', 'kicker']

    def play(self, actions):
        """Given (action1,action2) returns (resulting_state, (rewward1, reward2))
        resulting state is 's'
        """
        return ({('left', 'left'): (0.6, 0.4),
                ('left', 'right'): (0.3, 0.7),
                ('right', 'left'): (0.2, 0.8),
                ('right', 'right'): (0.9,0.1)
              }[actions], 's')

```

```python
class GameShow(Displayable):
    def __init__(self):
        self.num_agents = 2
        self.states = ['s']
        self.initial_state = 's'
        self.actions = [['takes', 'gives']]*2
        self.players = ['Agent 1', 'Agent 2']

    def play(self, actions):
        return ({('takes', 'takes'): (1, 1),
                 ('takes', 'gives'): (11, 0),
                 ('gives', 'takes'): (0, 11),
                 ('gives', 'gives'): (10, 10)
                }[actions], 's')


class UniqueNEGameExample(Displayable):
    def __init__(self):
        self.num_agents = 2
        self.states = ['s']
        self.initial_state = 's'
        self.actions = [['a1', 'b1', 'c1'],['d2', 'e2', 'f2']]
        self.players = ['agent 1 does', 'agent 2 does']

    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], 's')
```

masLearn.py — (continued)

```python
# Choose one:
# gm = ShoppingGame()
# gm = SoccerGame()
# gm = GameShow()
# gm = UniqueNEGameExample()

from rlQLearner import Q_learner
from rlProblem import RL_agent
# Choose one of the combinations of learners:
# sim=SimulateGame(gm,[StochasticPIAgent, StochasticPIAgent]);
    sim.go(10000)
# sim= SimulateGame(gm,[Q_learner, Q_learner]); sim.go(10000)
# sim=SimulateGame(gm,[Q_learner, StochasticPIAgent]); sim.go(10000)
```

```


# sim.plot_dynamics()

# empirical proportion that agents did their action at index 1:
# sim.action_dist([1,1])

# (unnormalized) empirical distribution for agent 0
# sim.agents[0].dist
```

Exercise 14.1 Try the game show game (prisoner's dilemma) with two `StochasticPIAgent` agents and `alpha_fun=lambda k:0.1`. Try also 0.01. Why does this work qualitatively different? Is this better?

Chapter 15

Relational Learning

15.1 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)
                       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)
```

Exercise 15.1 The above code updates the parameters after each example, but only regularizes after the whole batch. Change the program so that it implements stochastic gradient descent with a given batch size, and only updates the parameters after a batch.

Exercise 15.2 In the previous questions, can the regularization avoid iterating through the parameters for all users and items after a batch? Consider items that are in many batches versus those in a few or even no batches. (Warning: This is challenging to get right.)

15.1.1 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()
```

15.1.2 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.
        """
```

```python
        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)
```

15.2 Relational Probabilistic Models

The following implements relational belief networks – belief networks with plates. Plates correspond to logical variables.

relnProbModels.py — Relational Probabilistic Models: belief networks with plates

```python
from display import Displayable
from probGraphicalModels import BeliefNetwork
```

```
from variable import Variable
from probRC import ProbRC
from probFactors import Prob
import random

boolean = [False, True]
```

A ParVar is a parametrized random variable, which consists of the name, a list of logical variables (plates), a domain, and a position. For each assignment of an entity to each logical variable, there is a random variable in a grounding.

relnProbModels.py — (continued)

```
class ParVar(object):
    """Parametrized random variable"""
    def __init__(self, name, log_vars, domain, position=None):
        self.name = name # string
        self.log_vars = log_vars
        self.domain = domain # list of values
        self.position = position if position else (random.random(), random.random())
        self.size = len(domain)
```

The class RBN is of relational belief networks.

relnProbModels.py — (continued)

```
class RBN(Displayable):
    def __init__(self, title, parvars, parfactors):
        self.title = title
        self.parvars = parvars
        self.parfactors = parfactors
        self.log_vars = {V for PV in parvars for V in PV.log_vars}

    def ground(self, populations):
        """Ground the belief network with the populations of the logical variables.
        populations is a dictionary that maps each logical variable to the list of individuals.
        Returns a belief network representation of the grounding.
        """
        assert all(lv in populations for lv in self.log_vars)
        self.cps = []   # conditional probabilities in the grounding
        self.var_dict = {}  # ground variables created
        for pp in self.parfactors:
           self.ground_parfactor(pp, list(self.log_vars), populations, {})
        return BeliefNetwork(self.title+"_grounded", self.var_dict.values(), self.cps)

    def ground_parfactor(self, parfactor, lvs, populations, context):
        """
        parfactor is the parfactor to get instances of
        lvs is a list of the logical variables in parfactor not assigned in context
```

```
        populations is {logical_variable: population} dictionary
        context is a {logical_variable:value} dictionary for
            logical_variable in parfactor
        """
        if lvs == []:
            if isinstance(parfactor, Prob):
                self.cps.append(Prob(self.ground_pvr(parfactor.child,context),
                                     [self.ground_pvr(p,context) for p in
                                         parfactor.parents],
                                     parfactor.values))
            else:
                print("Parfactor not implemented for",parfactor,"of
                    type",type(parfactor))
        else:
            for val in populations[lvs[0]]:
                self.ground_parfactor(parfactor, lvs[1:], populations,
                    {lvs[0]:val}|context)


    def ground_pvr(self, prv, context):
        """grounds a parametrized random variable with respect to a context
        prv is a parametrized random variable
        context is a logical_variable:value dictionary that assigns all
            logical variables in prv
        """
        if isinstance(prv,ParVar):
            args = tuple(context[lv] for lv in prv.log_vars)
            if (prv,args) in self.var_dict:
                return self.var_dict[(prv,args)]
            else:
                new_gv = GrVar(prv, args)
                self.var_dict[(prv,args)] = new_gv
                return new_gv
        else: # allows for non-parametrized random variables
            return prv
```

A GrVar is a variable constructed by grounding a parametrized random variable with respect to a tuple of values for the logical variables.

relnProbModels.py — (continued)

```
class GrVar(Variable):
    """Grounded Variable"""
    def __init__(self,parvar,args):
        (x,y) = parvar.position
        pos = (x + random.uniform(-0.2,0.2), y + random.uniform(-0.2,0.2))
        Variable.__init__(self,parvar.name+"("+",".join(args)+")",
            parvar.domain, pos)
        self.parvar= parvar
        self.args = tuple(args)
        self.hash_value = None

```

```
    def __hash__(self):
        if self.hash_value is None:
            self.hash_value = hash((self.parvar, self.args))
        return self.hash_value

    def __eq__(self, other):
        return isinstance(other,GrVar) and self.parvar == other.parvar and
            self.args == other.args
```

The following is a representation of Examples 17.5-17.7 of Poole and Mackworth [2023]. The plate model – represented here using `grades` – is shown in Figure 17.4. The observation in `obs` corresponds to the dataset of Figure 17.3. The grounding in `grades_gr` corresponds to Figure 17.5, but also includes the Grade variables no needed to answer the query (see exercise below).

Try the commented out queries to the Python shell:

relnProbModels.py — (continued)

```
Int = ParVar("Intelligent", ["St"], boolean, position=(0.25,0.75))
Grade = ParVar("Grade", ["St","Co"], ["A", "B", "C"], position=(0.5,0.25))
Diff = ParVar("Difficult", ["Co"], boolean, position=(0.75,0.75))

pg = Prob(Grade, [Int, Diff],
               [[{"A": 0.1, "B":0.4, "C":0.5},
                    {"A": 0.01, "B":0.09, "C":0.9}],
                [{"A": 0.9, "B":0.09, "C":0.01},
                     {"A": 0.5, "B":0.4, "C":0.1}]])
pi = Prob( Int, [], [0.5, 0.5])
pd = Prob( Diff, [], [0.5, 0.5])
grades = RBN("Grades RBN", {Int, Grade, Diff}, {pg,pi,pd})


#grades_gr = grades.ground({"St":["s1", "s2", "s3", "s4"], "Co":["c1",
    "c2", "c3", "c4"]})

obs = {GrVar(Grade,["s1","c1"]):"A", GrVar(Grade,["s2","c1"]):"C",
    GrVar(Grade,["s1","c2"]):"B",
          GrVar(Grade,["s2","c3"]):"B", GrVar(Grade,["s3","c2"]):"B",
              GrVar(Grade,["s4","c3"]):"B"}

# grades_rc = ProbRC(grades_gr)
# grades_rc.query(GrVar(Grade,["s3","c4"]), obs)
# grades_rc.query(GrVar(Grade,["s4","c4"]), obs)
# grades_rc.query(GrVar(Int,["s3"]), obs)
# grades_rc.query(GrVar(Int,["s4"]), obs)
```

Exercise 15.3 The grounding above creates a random variable for each element for each possible combination of individuals in the populations. Change it so that it only creates as many random variables as needed to answer a query. For example, for the observations and queries above, only the variables in Figure 17.5 need to be created.

Exercise 15.4 Displaying the ground network, e.g., using `grades_gr.show()`, creates a messy diagram. Make it so that the user can provide offsets for each individual and uses the position of the prv plus the offsets of all the individuals involved. Use this create to create a 2D grid of grades in the example above.

Chapter 16

Version History

- 2023-07-31 Version 0.9.7 includes relational probabilistic models and smaller changes
- 2023-06-06 Version 0.9.6 controllers are more consistent. Many smaller changes.
- 2022-08-13 Version 0.9.5 major revisions including extra code for causality and deep learning
- 2021-07-08 Version 0.9.1 updated the CSP code to have the same representation of variables as used by the probability code
- 2021-05-13 Version 0.9.0 Major revisions to chapters 8 and 9. Introduced recursive conditioning, simplified much code. New section on multiagent reinforcement learning.
- 2020-11-04 Version 0.8.6 simplified value iteration for MDPs.
- 2020-10-20 Version 0.8.4 planning simplified, and 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

Chen, T. and Guestrin, C. (2016), Xgboost: A scalable tree boosting system. In *KDD '16: 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, URL `https://doi.org/10.1145/2939672.2939785`. 168

Chollet, F. (2021), *Deeep Learning with Python*. Manning. 171

Dua, D. and Graff, C. (2017), UCI machine learning repository. URL `http://archive.ics.uci.edu/ml`. 133

Glorot, X. and Bengio, Y. (2010), Understanding the difficulty of training deep feedforward neural networks. In *Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, URL `https://proceedings.mlr.press/v9/glorot10a.html`. 172

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017), LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems 30*. 168

Lichman, M. (2013), UCI machine learning repository. URL `http://archive.ics.uci.edu/ml`. 133

Pérez, F. and Granger, B. E. (2007), IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, URL `https://ipython.org`. 10

Poole, D. L. and Mackworth, A. K. (2023), *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 3rd edition, URL `https://artint.info`. 9, 23, 25, 45, 46, 62, 107, 193, 199, 201, 278, 279, 284, 288, 318

Index