

Python code for Artificial Intelligence: Foundations of Computational Agents

David L. Poole and Alan K. Mackworth

Version 0.9.6 of June 29, 2023.

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

©David L Poole and Alan K Mackworth 2017-2023.

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

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

The authors and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Contents

Contents	3
1 Python for Artificial Intelligence	9
1.1 Why Python?	9
1.2 Getting Python	10
1.3 Running Python	10
1.4 Pitfalls	11
1.5 Features of Python	12
1.5.1 f-strings	12
1.5.2 Lists, Tuples, Sets, Dictionaries and Comprehensions . .	12
1.5.3 Functions as first-class objects	13
1.5.4 Generators	14
1.6 Useful Libraries	16
1.6.1 Timing Code	16
1.6.2 Plotting: Matplotlib	16
1.7 Utilities	17
1.7.1 Display	17
1.7.2 Argmax	19
1.7.3 Probability	20
1.7.4 Dictionary Union	20
1.8 Testing Code	21
2 Agent Architectures and Hierarchical Control	23
2.1 Representing Agents and Environments	23
2.2 Paper buying agent and environment	25
2.2.1 The Environment	25

2.2.2	The Agent	27
2.2.3	Plotting	27
2.3	Hierarchical Controller	28
2.3.1	Environment	29
2.3.2	Body	29
2.3.3	Middle Layer	32
2.3.4	Top Layer	33
2.3.5	Plotting	34
3	Searching for Solutions	37
3.1	Representing Search Problems	37
3.1.1	Explicit Representation of Search Graph	38
3.1.2	Paths	40
3.1.3	Example Search Problems	41
3.2	Generic Searcher and Variants	45
3.2.1	Searcher	45
3.2.2	Frontier as a Priority Queue	46
3.2.3	A* Search	48
3.2.4	Multiple Path Pruning	49
3.3	Branch-and-bound Search	51
4	Reasoning with Constraints	55
4.1	Constraint Satisfaction Problems	55
4.1.1	Variables	55
4.1.2	Constraints	56
4.1.3	CSPs	57
4.1.4	Examples	59
4.2	A Simple Depth-first Solver	68
4.3	Converting CSPs to Search Problems	69
4.4	Consistency Algorithms	71
4.4.1	Direct Implementation of Domain Splitting	74
4.4.2	Domain Splitting as an interface to graph searching	76
4.5	Solving CSPs using Stochastic Local Search	77
4.5.1	Any-conflict	80
4.5.2	Two-Stage Choice	81
4.5.3	Updatable Priority Queues	83
4.5.4	Plotting Run-Time Distributions	85
4.5.5	Testing	86
4.6	Discrete Optimization	87
4.6.1	Branch-and-bound Search	88
5	Propositions and Inference	91
5.1	Representing Knowledge Bases	91
5.2	Bottom-up Proofs (with askables)	94
5.3	Top-down Proofs (with askables)	96

5.4	Debugging and Explanation	97
5.5	Assumables	101
5.6	Negation-as-failure	104
6	Deterministic Planning	107
6.1	Representing Actions and Planning Problems	107
6.1.1	Robot Delivery Domain	108
6.1.2	Blocks World	110
6.2	Forward Planning	112
6.2.1	Defining Heuristics for a Planner	115
6.3	Regression Planning	117
6.3.1	Defining Heuristics for a Regression Planner	119
6.4	Planning as a CSP	120
6.5	Partial-Order Planning	123
7	Supervised Machine Learning	131
7.1	Representations of Data and Predictions	132
7.1.1	Creating Boolean Conditions from Features	135
7.1.2	Evaluating Predictions	137
7.1.3	Creating Test and Training Sets	138
7.1.4	Importing Data From File	139
7.1.5	Augmented Features	142
7.2	Generic Learner Interface	144
7.3	Learning With No Input Features	145
7.3.1	Evaluation	147
7.4	Decision Tree Learning	149
7.5	Cross Validation and Parameter Tuning	153
7.6	Linear Regression and Classification	157
7.7	Boosting	163
7.7.1	Gradient Tree Boosting	166
8	Neural Networks and Deep Learning	169
8.1	Layers	169
8.2	Feedforward Networks	172
8.3	Improved Optimization	174
8.3.1	Momentum	174
8.3.2	RMS-Prop	175
8.4	Dropout	176
8.4.1	Examples	177
9	Reasoning with Uncertainty	183
9.1	Representing Probabilistic Models	183
9.2	Representing Factors	183
9.3	Conditional Probability Distributions	185
9.3.1	Logistic Regression	185

9.3.2	Noisy-or	186
9.3.3	Tabular Factors	187
9.4	Graphical Models	188
9.4.1	Example Belief Networks	190
9.5	Inference Methods	195
9.6	Naive Search	197
9.7	Recursive Conditioning	198
9.8	Variable Elimination	202
9.9	Stochastic Simulation	206
9.9.1	Sampling from a discrete distribution	206
9.9.2	Sampling Methods for Belief Network Inference	207
9.9.3	Rejection Sampling	208
9.9.4	Likelihood Weighting	209
9.9.5	Particle Filtering	210
9.9.6	Examples	211
9.9.7	Gibbs Sampling	213
9.9.8	Plotting Behaviour of Stochastic Simulators	214
9.10	Hidden Markov Models	216
9.10.1	Exact Filtering for HMMs	218
9.10.2	Localization	220
9.10.3	Particle Filtering for HMMs	222
9.10.4	Generating Examples	224
9.11	Dynamic Belief Networks	225
9.11.1	Representing Dynamic Belief Networks	225
9.11.2	Unrolling DBNs	229
9.11.3	DBN Filtering	229
10	Learning with Uncertainty	233
10.1	K-means	233
10.2	EM	237
11	Causality	243
11.1	Do Questions	243
11.2	Counterfactual Example	245
12	Planning with Uncertainty	249
12.1	Decision Networks	249
12.1.1	Example Decision Networks	251
12.1.2	Recursive Conditioning for decision networks	256
12.1.3	Variable elimination for decision networks	260
12.2	Markov Decision Processes	262
12.2.1	Value Iteration	265
12.2.2	Showing Grid MDPs	266
12.2.3	Asynchronous Value Iteration	268

13 Reinforcement Learning	273
13.1 Representing Agents and Environments	273
13.1.1 Simulating an environment from an MDP	276
13.1.2 Monster Game	277
13.2 Q Learning	279
13.2.1 Exploration Strategies	281
13.2.2 Testing Q-learning	282
13.3 Q-learning with Experience Replay	283
13.4 Model-based Reinforcement Learner	285
13.5 Reinforcement Learning with Features	288
13.5.1 Representing Features	288
13.5.2 Feature-based RL learner	291
14 Multiagent Systems	295
14.1 Minimax	295
14.1.1 Creating a two-player game	295
14.1.2 Minimax and α - β Pruning	298
14.2 Multiagent Learning	300
15 Relational Learning	307
15.1 Collaborative Filtering	307
15.1.1 Alternative Formulation	310
15.1.2 Plotting	310
15.1.3 Creating Rating Sets	311
16 Version History	315
Bibliography	317
Index	319

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 later¹ (<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 *pip3*. You can install matplotlib using

```
pip3 install matplotlib
```

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

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

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

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

We recommend using the enhanced interactive python **ipython** (<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
```

¹The 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 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

³Numbered 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
11 fun_list1 = []
12 for i in range(5):
13     def fun1(e):
14         return e+i
15     fun_list1.append(fun1)
16
17 fun_list2 = []
18 for i in range(5):
19     def fun2(e,iv=i):
20         return e+iv
21     fun_list2.append(fun2)
22
23 fun_list3 = [lambda e: e+i for i in range(5)]
24
25 fun_list4 = [lambda e,iv=i: e+iv for i in range(5)]
26
27 i=56

```

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

```

pythonDemo.py — (continued)
29 # in Shell do
30 ## ipython -i pythonDemo.py
31 # Try these (copy text after the comment symbol and paste in the Python
    prompt):
32 # print([f(10) for f in fun_list1])
33 # print([f(10) for f in fun_list2])
34 # print([f(10) for f in fun_list3])
35 # print([f(10) for f in fun_list4])

```

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

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

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)
37 def myrange(start, stop, step=1):
38     """enumerates the values from start in steps of size step that are
39     less than stop.
40     """
41     assert step>0, f"only positive steps implemented in myrange: {step}"
42     i = start
43     while i<stop:
44         yield i
45         i += step
46
47 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)
49 def ga(n):
50     """generates square of even nonnegative integers less than n"""
51     for e in range(n):
52         if e%2==0:
53             yield e*e
54 a = ga(20)
```

The sequence of `next(a)`, and `list(a)` gives exactly the same results as the comprehension in Section 1.5.2.

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

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

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

1.6 Useful Libraries

1.6.1 Timing Code

In order to compare algorithms, we often want to compute how long a program takes; this is called the **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 *timeit* a few (say 3) times. When reporting the time of any computation, you should be explicit and explain what you are reporting. Usually the minimum time is the one to report.

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)
60 import matplotlib.pyplot as plt
61
62 def myplot(minv,maxv,step,fun1,fun2):
63     plt.ion() # make it interactive
64     plt.xlabel("The x axis")
65     plt.ylabel("The y axis")
66     plt.xscale('linear') # Makes a 'log' or 'linear' scale
67     xvalues = range(minv,maxv,step)
68     plt.plot(xvalues,[fun1(x) for x in xvalues],
69              label="The first fun")
70     plt.plot(xvalues,[fun2(x) for x in xvalues], linestyle='--',color='k',
71              label=fun2.__doc__) # use the doc string of the function
72     plt.legend(loc="upper right") # display the legend
73
74 def slin(x):
75     """y=2x+7"""
76     return 2*x+7
77 def sqfun(x):
78     """y=(x-40)^2/10-20"""
79     return (x-40)**2/10-20
80
81 # Try the following:
82 # from pythonDemo import myplot, slin, sqfun
83 # import matplotlib.pyplot as plt
84 # myplot(0,100,1,slin,sqfun)
85 # plt.legend(loc="best")
86 # import math
87 # plt.plot([41+40*math.cos(th/10) for th in range(50)],
88 #          [100+100*math.sin(th/10) for th in range(50)])
89 # plt.text(40,100,"ellipse?")
90 # plt.xscale('log')

```

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

1.7 Utilities

1.7.1 Display

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

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

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

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

The definition of *display* is:

```

_____display.py — A simple way to trace the intermediate steps of algorithms. _____
11 class Displayable(object):
12     """Class that uses 'display'.
13     The amount of detail is controlled by max_display_level
14     """
15     max_display_level = 1 # can be overridden in subclasses or instances
16
17     def display(self, level, *args, **nargs):
18         """print the arguments if level is less than or equal to the
19         current max_display_level.
20         level is an integer.
21         the other arguments are whatever arguments print can take.
22         """
23         if level <= self.max_display_level:
24             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)

```

26 def visualize(func):
27     """A decorator for algorithms that do interactive visualization.
28     Ignored here.
29     """
30     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 of 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

```

11 import random
12 import math
13
14 def argmaxall(gen):
15     """gen is a generator of (element,value) pairs, where value is a real.
16     argmaxall returns a list of all of the elements with maximal value.
17     """
18     maxv = -math.inf    # negative infinity
19     maxvals = []       # list of maximal elements
20     for (e,v) in gen:
21         if v>maxv:
22             maxvals,maxv = [e], v
23         elif v==maxv:
24             maxvals.append(e)
25     return maxvals
26
27 def argmaxe(gen):
28     """gen is a generator of (element,value) pairs, where value is a real.
29     argmaxe returns an element with maximal value.
30     If there are multiple elements with the max value, one is returned at
31     random.
32     """
33     return random.choice(argmaxall(gen))
34
35 def argmax(lst):
36     """returns maximum index in a list"""
37     return argmaxe(enumerate(lst))
38
39 # Try:
40 # argmax([1,6,3,77,3,55,23])
41
42 def argmaxd(dct):
43     """returns the arg max of a dictionary dct"""

```

```

42 |     return argmaxe(dct.items())
43 | # Try:
44 | # arxmaxd({2:5,5:9,7:7})

```

Exercise 1.3 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) _____
45 | def flip(prob):
46 |     """return true with probability prob"""
47 |     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) _____
49 | def pick_from_dist(item_prob_dist):
50 |     """ returns a value from a distribution.
51 |     item_prob_dist is an item:probability dictionary, where the
52 |     probabilities sum to 1.
53 |     returns an item chosen in proportion to its probability
54 |     """
55 |     ranreal = random.random()
56 |     for (it,prob) in item_prob_dist.items():
57 |         if ranreal < prob:
58 |             return it
59 |         else:
60 |             ranreal -= prob
61 |     raise RuntimeError(f"{item_prob_dist} is not a probability
    distribution")

```

1.7.4 Dictionary Union

This is now | in Python 3.9, has been 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) _____
63 | def dict_union(d1,d2):
64 |     """returns a dictionary that contains the keys of d1 and d2.

```

```
65 |     The value for each key that is in d2 is the value from d2,  
66 |     otherwise it is the value from d1.  
67 |     This does not have side effects.  
68 |     """  
69 |     d = dict(d1) # copy d1  
70 |     d.update(d2)  
71 |     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) _____  
73 | def test():  
74 |     """Test part of utilities"""  
75 |     assert argmax(enumerate([1,6,55,3,55,23])) in [2,4]  
76 |     assert dict_union({1:4, 2:5, 3:4},{5:7, 2:9}) == {1:4, 2:9, 3:4, 5:7}  
77 |     print("Passed unit test in utilities")  
78 |  
79 | if __name__ == "__main__":  
80 |     test()
```


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 environment are treated as objects in the sense 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 its internal state and returns the next percept, using the method `do`.

An **agent** takes the percept, updates its internal state, and outputs its 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 273).
- 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
11 from display import Displayable
12
13 class Agent(Displayable):
14
15     def initial_action(self, percept):
16         """return the initial action"""
17         raise NotImplementedError("go") # abstract method
18
19     def select_action(self, percept):
20         """return the next action (and update internal state) given percept
21         percept is variable:value dictionary
22         """
23         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)
25 class Environment(Displayable):
26     def initial_percept(self):
27         """returns the initial percept for the agent"""
28         raise NotImplementedError("initial_percept") # abstract method
29
30     def do(self, action):
31         """does the action in the environment

```



```

32         returns the next percept """
33         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)
35 class Simulate(Displayable):
36     """simulate the interaction between the agent and the environment
37     for n time steps.
38     Returns a pair of the agent state and the environment state.
39     """
40     def __init__(self, agent, environment):
41         self.agent = agent
42         self.env = environment
43         self.percept = self.env.initial_percept()
44         self.percept_history = [self.percept]
45         self.action_history = []
46
47     def go(self, n):
48         for i in range(n):
49             action = self.agent.select_action(self.percept)
50             self.display(2, f"i={i} action={action}")
51             self.percept = self.env.do(action)
52             self.display(2, f"    percept={self.percept}")

```

2.2 Paper buying agent and environment

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

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

2.2.1 The Environment

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

Here we assume that the prices are obtained from the *prices* list (which cycles) plus a random integer in range $[0, \text{max_price_addon})$ plus a linear "infla-

tion". The agent cannot access the price model; it just observes the prices and the amount in stock.

```

agentBuying.py — Paper-buying agent
11 import random
12 from agents import Agent, Environment, Simulate
13 from utilities import pick_from_dist
14
15 class TP_env(Environment):
16     prices = [234, 234, 234, 234, 255, 255, 275, 275, 211, 211, 211,
17             234, 234, 234, 234, 199, 199, 275, 275, 234, 234, 234, 234, 255,
18             255, 260, 260, 265, 265, 265, 265, 270, 270, 255, 255, 260, 260,
19             265, 265, 150, 150, 265, 265, 270, 270, 255, 255, 260, 260, 265,
20             265, 265, 265, 270, 270, 211, 211, 255, 255, 260, 260, 265, 265,
21             260, 265, 270, 270, 205, 255, 255, 260, 260, 265, 265, 265, 265,
22             270, 270]
23     max_price_addon = 20 # maximum of random value added to get price
24
25     def __init__(self):
26         """paper buying agent"""
27         self.time=0
28         self.stock=20
29         self.stock_history = [] # memory of the stock history
30         self.price_history = [] # memory of the price history
31
32     def initial_percept(self):
33         """return initial percept"""
34         self.stock_history.append(self.stock)
35         price = self.prices[0]+random.randrange(self.max_price_addon)
36         self.price_history.append(price)
37         return {'price': price,
38               'instock': self.stock}
39
40     def do(self, action):
41         """does action (buy) and returns percept consisting of price and
42            instock"""
43         used = pick_from_dist({6:0.1, 5:0.1, 4:0.1, 3:0.3, 2:0.2, 1:0.2})
44         # used = pick_from_dist({7:0.1, 6:0.2, 5:0.2, 4:0.3, 3:0.1, 2:0.1})
45         # uses more paper
46         bought = action['buy']
47         self.stock = self.stock+bought-used
48         self.stock_history.append(self.stock)
49         self.time += 1
50         price = (self.prices[self.time%len(self.prices)] # repeating pattern
51                 +random.randrange(self.max_price_addon) # plus randomness
52                 +self.time//2) # plus inflation
53         self.price_history.append(price)
54         return {'price': price,
55               '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)
55 class TP_agent(Agent):
56     def __init__(self):
57         self.spent = 0
58         percept = env.initial_percept()
59         self.ave = self.last_price = percept['price']
60         self.instock = percept['instock']
61         self.buy_history = []
62
63     def select_action(self, percept):
64         """return next action to carry out
65         """
66         self.last_price = percept['price']
67         self.ave = self.ave+(self.last_price-self.ave)*0.05
68         self.instock = percept['instock']
69         if self.last_price < 0.9*self.ave and self.instock < 60:
70             tobuy = 48
71         elif self.instock < 12:
72             tobuy = 12
73         else:
74             tobuy = 0
75         self.spent += tobuy*self.last_price
76         self.buy_history.append(tobuy)
77         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)
79 env = TP_env()
80 ag = TP_agent()
81 sim = Simulate(ag,env)
82 #sim.go(90)
83 #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)
85 import matplotlib.pyplot as plt
86
87 class Plot_history(object):

```

```

88     """Set up the plot for history of price and number in stock"""
89     def __init__(self, ag, env):
90         self.ag = ag
91         self.env = env
92         plt.ion()
93         plt.xlabel("Time")
94         plt.ylabel("Value")
95
96
97     def plot_env_hist(self):
98         """plot history of price and instock"""
99         num = len(env.stock_history)
100         plt.plot(range(num), env.price_history, label="Price")
101         plt.plot(range(num), env.stock_history, label="In stock")
102         plt.legend()
103         #plt.draw()
104
105     def plot_agent_hist(self):
106         """plot history of buying"""
107         num = len(ag.buy_history)
108         plt.bar(range(1, num+1), ag.buy_history, label="Bought")
109         plt.legend()
110         #plt.draw()
111
112 # pl = Plot_history(ag, env)
113 # sim.go(90)
114 #pl.plot_env_hist()
115 #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.

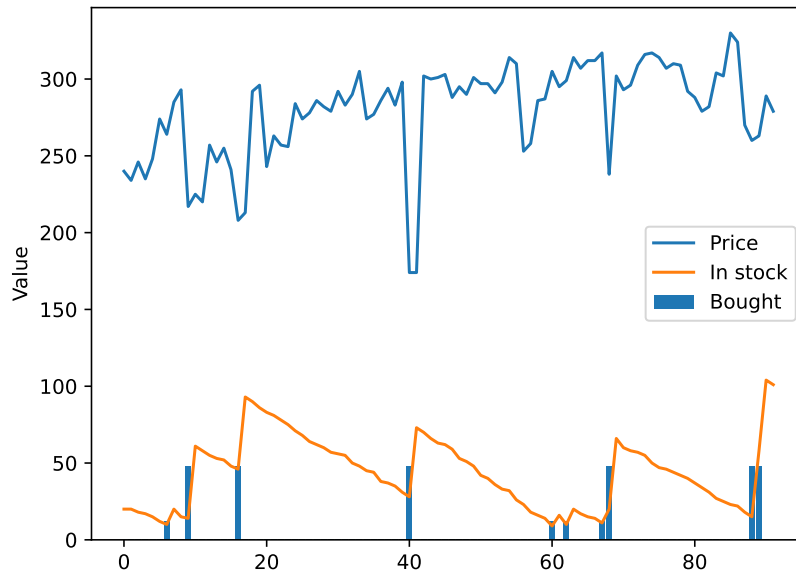


Figure 2.1: Percept and command traces for the paper-buying agent

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
11 import math
12 from agents import Environment
13
14 class Rob_env(Environment):
15     def __init__(self, walls = {}):
16         """walls is a set of line segments
17            where each line segment is of the form ((x0,y0),(x1,y1))
18            """
19         self.walls = walls

```

2.3.2 Body

The body defines everything about the agent body.

```

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

```

```

64         if self.plotting:
65             plt.plot([self.rob_x],[self.rob_y],"r*",markersize=20.0)
66             plt.draw()
67         self.rob_x, self.rob_y = rob_x_new, rob_y_new
68         self.history.append((self.rob_x, self.rob_y))
69         if self.plotting and not self.crashed:
70             plt.plot([self.rob_x],[self.rob_y],"go")
71             plt.draw()
72             plt.pause(self.sleep_time)
73         return self.percept()

```

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

agentEnv.py — (continued)

```

75     def whisker(self):
76         """returns true whenever the whisker sensor intersects with a wall
77         """
78         whisk_ang_world = (self.rob_dir-self.whisker_angle)*math.pi/180
79         # angle in radians in world coordinates
80         wx = self.rob_x + self.whisker_length * math.cos(whisk_ang_world)
81         wy = self.rob_y + self.whisker_length * math.sin(whisk_ang_world)
82         whisker_line = ((self.rob_x,self.rob_y),(wx,wy))
83         hit = any(line_segments_intersect(whisker_line,wall)
84                 for wall in self.env.walls)
85         if hit:
86             self.wall_history.append((self.rob_x, self.rob_y))
87             if self.plotting:
88                 plt.plot([self.rob_x],[self.rob_y],"ro")
89                 plt.draw()
90         return hit
91
92     def line_segments_intersect(linea,lineb):
93         """returns true if the line segments, linea and lineb intersect.
94         A line segment is represented as a pair of points.
95         A point is represented as a (x,y) pair.
96         """
97         ((x0a,y0a),(x1a,y1a)) = linea
98         ((x0b,y0b),(x1b,y1b)) = lineb
99         da, db = x1a-x0a, x1b-x0b
100        ea, eb = y1a-y0a, y1b-y0b
101        denom = db*ea-eb*da
102        if denom==0: # line segments are parallel
103            return False
104        cb = (da*(y0b-y0a)-ea*(x0b-x0a))/denom # position along line b
105        if cb<0 or cb>1:
106            return False
107        ca = (db*(y0b-y0a)-eb*(x0b-x0a))/denom # position along line a
108        return 0<=ca<=1
109
110 # Test cases:

```

```

111 # assert line_segments_intersect(((0,0),(1,1)),((1,0),(0,1)))
112 # assert not line_segments_intersect(((0,0),(1,1)),((1,0),(0.6,0.4)))
113 # assert line_segments_intersect(((0,0),(1,1)),((1,0),(0.4,0.6)))

```

2.3.3 Middle Layer

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

```

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

```

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


```

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

```

2.3.4 Top Layer

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

```

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

```

```

25
26     def do(self, plan):
27         """carry out actions.
28         actions is of the form {'visit':list_of_locations}
29         It visits the locations in turn.
30         """
31         to_do = plan['visit']
32         for loc in to_do:
33             position = self.locations[loc]
34             arrived = self.middle.do({'go_to':position,
35                                     'timeout':self.timeout})
36             self.display(1, "Arrived at", loc, arrived)

```

2.3.5 Plotting

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

```

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

```

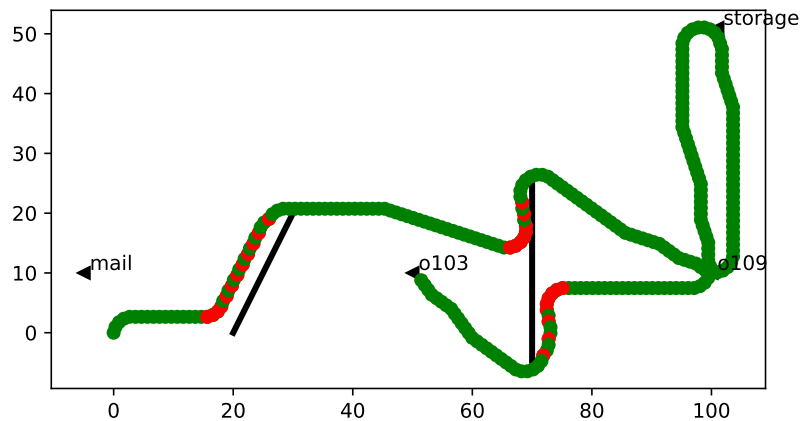


Figure 2.2: A trace of the trajectory of the agent. Red dots correspond to the whisker sensor being on; the green dot to the whisker sensor being off. The agent starts at position $(0,0)$ facing up.

```
64 | plt.plot(wxs,wys,"ro")
65 | #plt.draw()
```

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

Exercise 2.2 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)
```

```
81 # Robot Trap for which the current controller cannot escape:
82 trap_env = Rob_env({((10,-21),(10,0)), ((10,10),(10,31)),
      ((30,-10),(30,0)),
83      ((30,10),(30,20)), ((50,-21),(50,31)),
      ((10,-21),(50,-21)),
```

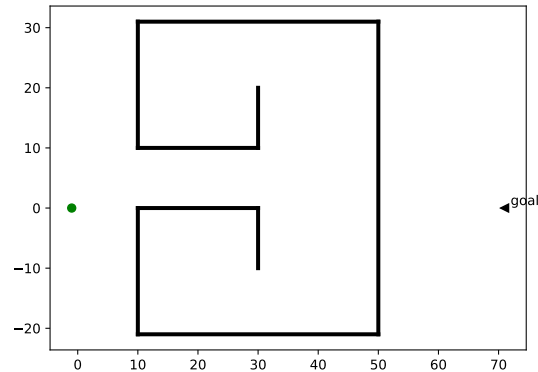


Figure 2.3: Robot trap

```

84 |         ((10,0),(30,0)), ((10,10),(30,10)), ((10,31),(50,31)))
85 | trap_body = Rob_body(trap_env,init_pos=(-1,0,90))
86 | trap_middle = Rob_middle_layer(trap_body)
87 | trap_top = Rob_top_layer(trap_middle,locations={'goal':(71,0)})
88 |
89 | # Robot trap exercise:
90 | # pl=Plot_env(trap_body,trap_top)
91 | # trap_top.do({'visit':['goal']})

```

Searching for Solutions

3.1 Representing Search Problems

A search problem consists of:

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

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

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

```
searchProblem.py — representations of search problems
11 class Search_problem(object):
12     """A search problem consists of:
13     * a start node
14     * a neighbors function that gives the neighbors of a node
15     * a specification of a goal
16     * a (optional) heuristic function.
```

```

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

```

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

searchProblem.py — (continued)

```

36 class Arc(object):
37     """An arc has a from_node and a to_node node and a (non-negative)
38     cost"""
39     def __init__(self, from_node, to_node, cost=1, action=None):
40         self.from_node = from_node
41         self.to_node = to_node
42         self.action = action
43         self.cost=cost
44         assert cost >= 0, (f"Cost cannot be negative: {self}, cost={cost}")
45
46     def __repr__(self):
47         """string representation of an arc"""
48         if self.action:
49             return f"{self.from_node} --{self.action}--> {self.to_node}"
50         else:
51             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.

```

52 class Search_problem_from_explicit_graph(Search_problem):
53     """A search problem consists of:
54     * a list or set of nodes
55     * a list or set of arcs
56     * a start node
57     * a list or set of goal nodes
58     * a dictionary that maps each node into its heuristic value.
59     * a dictionary that maps each node into its (x,y) position
60     """
61
62     def __init__(self, nodes, arcs, start=None, goals=set(), hmap={},
63               positions={}):
64         self.neighs = {}
65         self.nodes = nodes
66         for node in nodes:
67             self.neighs[node]=[]
68         self.arcs = arcs
69         for arc in arcs:
70             self.neighs[arc.from_node].append(arc)
71         self.start = start
72         self.goals = goals
73         self.hmap = hmap
74         self.positions = positions
75
76     def start_node(self):
77         """returns start node"""
78         return self.start
79
80     def is_goal(self,node):
81         """is True if node is a goal"""
82         return node in self.goals
83
84     def neighbors(self,node):
85         """returns the neighbors of node (a list of arcs)"""
86         return self.neighs[node]
87
88     def heuristic(self,node):
89         """Gives the heuristic value of node n.

```

```

89     Returns 0 if not overridden in the hmap."""
90     if node in self.hmap:
91         return self.hmap[node]
92     else:
93         return 0
94
95     def __repr__(self):
96         """returns a string representation of the search problem"""
97         res=""
98         for arc in self.arcs:
99             res += f"{arc}. "
100        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. Note that we only use the most basic form of Python's yield for enumerations (Section 1.5.4).

```

searchProblem.py — (continued)
102 class Path(object):
103     """A path is either a node or a path followed by an arc"""
104
105     def __init__(self, initial, arc=None):
106         """initial is either a node (in which case arc is None) or
107         a path (in which case arc is an object of type Arc)"""
108         self.initial = initial
109         self.arc=arc
110         if arc is None:
111             self.cost=0
112         else:
113             self.cost = initial.cost+arc.cost
114
115     def end(self):
116         """returns the node at the end of the path"""

```



```

117         if self.arc is None:
118             return self.initial
119         else:
120             return self.arc.to_node
121
122     def nodes(self):
123         """enumerates the nodes for the path.
124         This enumerates the nodes in the path from the last elements
125         backwards.
126         """
127         current = self
128         while current.arc is not None:
129             yield current.arc.to_node
130             current = current.initial
131         yield current.initial
132
133     def initial_nodes(self):
134         """enumerates the nodes for the path before the end node.
135         This calls nodes() for the initial part of the path.
136         """
137         if self.arc is not None:
138             yield from self.initial.nodes()
139
140     def __repr__(self):
141         """returns a string representation of a path"""
142         if self.arc is None:
143             return str(self.initial)
144         elif self.arc.action:
145             return f"{self.initial}\n --{self.arc.action}-->
146                 {self.arc.to_node}"
147         else:
148             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)
148 problem1 = Search_problem_from_explicit_graph(
149     {'A', 'B', 'C', 'D', 'G'},
150     [Arc('A', 'B', 3), Arc('A', 'C', 1), Arc('B', 'D', 1), Arc('B', 'G', 3),
151       Arc('C', 'B', 1), Arc('C', 'D', 3), Arc('D', 'G', 1)],
152     start = 'A',
153     goals = {'G'},
154     positions={'A': (0, 2), 'B': (1, 1), 'C': (0, 1), 'D': (1, 0), 'G':
155               (2, 0)})

```

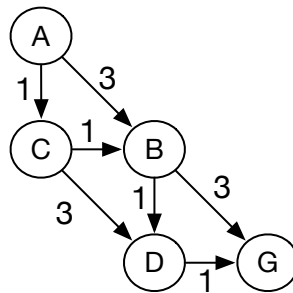


Figure 3.1: problem1

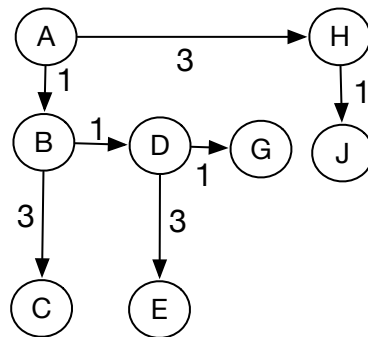


Figure 3.2: problem2

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

```

155 | searchProblem.py — (continued)
155 | problem2 = Search_problem_from_explicit_graph(
156 |     {'a','b','c','d','e','g','h','j'},
157 |     [Arc('a','b',1), Arc('b','c',3), Arc('b','d',1), Arc('d','e',3),
158 |       Arc('d','g',1), Arc('a','h',3), Arc('h','j',1)],
159 |     start = 'a',
160 |     goals = {'g'},
161 |     positions={'a': (0, 0), 'b': (0, 1), 'c': (0,4), 'd': (1,1), 'e': (1,4),
162 |               'g': (2,1), 'h': (3,0), 'j': (3,1)})

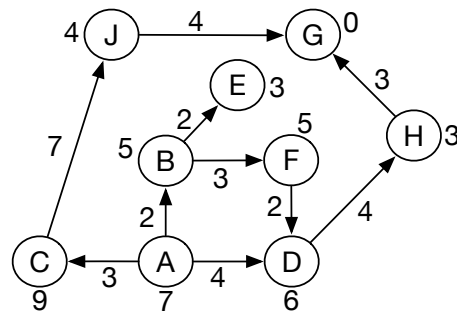
```

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.

```

164 | searchProblem.py — (continued)
164 | problem3 = Search_problem_from_explicit_graph(

```

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

```

165     {'a', 'b', 'c', 'd', 'e', 'g', 'h', 'j'},
166     [],
167     start = 'g',
168     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 Figure 3.13 of [Poole and Mackworth, 2023],

```

searchProblem.py — (continued)
170 simp_delivery_graph = Search_problem_from_explicit_graph(
171     {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J'},
172     [
173         Arc('A', 'B', 2),
174         Arc('A', 'C', 3),
175         Arc('A', 'D', 4),
176         Arc('B', 'E', 2),
177         Arc('B', 'F', 3),
178         Arc('C', 'J', 7),
179         Arc('D', 'H', 4),
180         Arc('F', 'D', 2),
181         Arc('H', 'G', 3),
182         Arc('J', 'G', 4)],
183     start = 'A',
184     goals = {'G'},
185     hmap = {
186         'A': 7,
187         'B': 5,
188         'C': 9,
189         'D': 6,
190         'E': 3,
191         'F': 5,
192         'G': 0,
193         'H': 3,
194         'J': 4,
195     })

```

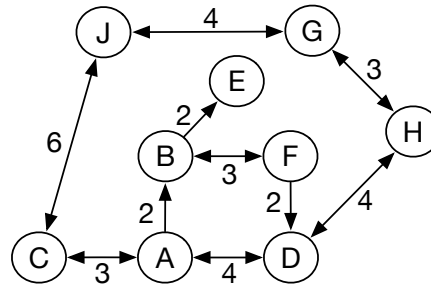


Figure 3.4: cyclic_simp_delivery_graph with arc costs

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.

```

195 searchProblem.py — (continued)
196 cyclic_simp_delivery_graph = Search_problem_from_explicit_graph(
197     {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J'},
198     [
199         Arc('A', 'B', 2),
200         Arc('A', 'C', 3),
201         Arc('A', 'D', 4),
202         Arc('B', 'A', 2),
203         Arc('B', 'E', 2),
204         Arc('B', 'F', 3),
205         Arc('C', 'A', 3),
206         Arc('C', 'J', 7),
207         Arc('D', 'A', 4),
208         Arc('D', 'H', 4),
209         Arc('F', 'B', 3),
210         Arc('F', 'D', 2),
211         Arc('G', 'H', 3),
212         Arc('G', 'J', 4),
213         Arc('H', 'D', 4),
214         Arc('H', 'G', 3),
215         Arc('J', 'C', 6),
216         Arc('J', 'G', 4)],
217     start = 'A',
218     goals = {'G'},
219     hmap = {
220         'A': 7,
221         'B': 5,
222         'C': 9,
223         'D': 6,
224         'E': 3,
225         'F': 5,
226         'G': 0,

```

```

225         'H': 3,
226         'J': 4,
227     })

```

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*
11 from display import Displayable, visualize
12
13 class Searcher(Displayable):
14     """returns a searcher for a problem.
15     Paths can be found by repeatedly calling search().
16     This does depth-first search unless overridden
17     """
18     def __init__(self, problem):
19         """creates a searcher from a problem
20         """
21         self.problem = problem
22         self.initialize_frontier()
23         self.num_expanded = 0
24         self.add_to_frontier(Path(problem.start_node()))
25         super().__init__()
26
27     def initialize_frontier(self):
28         self.frontier = []
29
30     def empty_frontier(self):
31         return self.frontier == []
32
33     def add_to_frontier(self, path):
34         self.frontier.append(path)
35
36     @visualize
37     def search(self):
38         """returns (next) path from the problem's start node
39         to a goal node.

```

```

40     Returns None if no path exists.
41     """
42     while not self.empty_frontier():
43         path = self.frontier.pop()
44         self.display(2, "Expanding:", path, "(cost:", path.cost, ")")
45         self.num_expanded += 1
46         if self.problem.is_goal(path.end()): # solution found
47             self.display(1, self.num_expanded, "paths have been expanded
48                 and",
49                     len(self.frontier), "paths remain in the
50                     frontier")
51             self.solution = path # store the solution found
52             return path
53         else:
54             neighs = self.problem.neighbors(path.end())
55             self.display(3, "Neighbors are", neighs)
56             for arc in reversed(list(neighs)):
57                 self.add_to_frontier(Path(path, arc))
58             self.display(3, "Frontier:", self.frontier)
59     self.display(1, "No (more) solutions. Total of",
60         self.num_expanded, "paths expanded.")

```

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

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

```

searchGeneric.py — (continued)
60 # Depth-first search for problem1; do the following:
61 # searcher1 = Searcher(searchProblem.problem1)
62 # searcher1.search() # find first solution
63 # searcher1.search() # find next solution (repeat until no solutions)
64 # searcher_sdg = Searcher(searchProblem.simp_delivery_graph)
65 # 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 elements were added to the queue, and the third element is the path that is on the queue. The use of the unique index ensures that the priority queue implementation does not compare paths; whether one path is less than another is not defined. It also lets us control what sort of search (e.g., depth-first or breadth-first) occurs when the value to be minimized does not give a unique next path.

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

```

searchGeneric.py — (continued)
67 import heapq      # part of the Python standard library
68 from searchProblem import Path
69
70 class FrontierPQ(object):
71     """A frontier consists of a priority queue (heap), frontierpq, of
72        (value, index, path) triples, where
73        * value is the value we want to minimize (e.g., path cost + h).
74        * index is a unique index for each element
75        * path is the path on the queue
76        Note that the priority queue always returns the smallest element.
77     """
78
79     def __init__(self):
80         """constructs the frontier, initially an empty priority queue
81         """
82         self.frontier_index = 0 # the number of items added to the frontier
83         self.frontierpq = [] # the frontier priority queue
84
85     def empty(self):
86         """is True if the priority queue is empty"""
87         return self.frontierpq == []
88
89     def add(self, path, value):
90         """add a path to the priority queue
91         value is the value to be minimized"""
92         self.frontier_index += 1 # get a new unique index
93         heapq.heappush(self.frontierpq, (value, -self.frontier_index, path))
94
95     def pop(self):
96         """returns and removes the path of the frontier with minimum value.
97         """
98         (_,_,path) = heapq.heappop(self.frontierpq)
99         return path

```

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

```

searchGeneric.py — (continued)

```

```

101     def count(self, val):
102         """returns the number of elements of the frontier with value=val"""
103         return sum(1 for e in self.frontierpq if e[0]==val)
104
105     def __repr__(self):
106         """string representation of the frontier"""
107         return str([(n,c,p) for (n,c,p) in self.frontierpq])
108
109     def __len__(self):
110         """length of the frontier"""
111         return len(self.frontierpq)
112
113     def __iter__(self):
114         """iterate through the paths in the frontier"""
115         for (_,_,path) in self.frontierpq:
116             yield path

```

3.2.3 A* Search

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

```

searchGeneric.py — (continued)
118 class AStarSearcher(Searcher):
119     """returns a searcher for a problem.
120     Paths can be found by repeatedly calling search().
121     """
122
123     def __init__(self, problem):
124         super().__init__(problem)
125
126     def initialize_frontier(self):
127         self.frontier = FrontierPQ()
128
129     def empty_frontier(self):
130         return self.frontier.empty()
131
132     def add_to_frontier(self, path):
133         """add path to the frontier with the appropriate cost"""
134         value = path.cost + self.problem.heuristic(path.end())
135         self.frontier.add(path, value)

```

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

```

searchGeneric.py — (continued)
137 import searchProblem as searchProblem
138
139 def test(SearchClass, problem=searchProblem.problem1,
140         solutions=[['G', 'D', 'B', 'C', 'A'] ]):
141     """Unit test for aipython searching algorithms.

```



```

141     SearchClass is a class that takes a problem and implements search()
142     problem is a search problem
143     solutions is a list of optimal solutions
144     """
145     print("Testing problem 1:")
146     schr1 = SearchClass(problem)
147     path1 = schr1.search()
148     print("Path found:", path1)
149     assert path1 is not None, "No path is found in problem1"
150     assert list(path1.nodes()) in solutions, "Shortest path not found in
151         problem1"
152     print("Passed unit test")
153
154 if __name__ == "__main__":
155     #test(Searcher)      # what needs to be changed to make this succeed?
156     test(AStarSearcher)
157
158 # example queries:
159 # searcher1 = Searcher(searchProblem.simp_delivery_graph) # DFS
160 # searcher1.search() # find first path
161 # searcher1.search() # find next path
162 # searcher2 = AStarSearcher(searchProblem.simp_delivery_graph) # A*
163 # searcher2.search() # find first path
164 # searcher2.search() # find next path
165 # searcher3 = Searcher(searchProblem.cyclic_simp_delivery_graph) # DFS
166 # searcher3.search() # find first path with DFS. What do you expect to
167     happen?
168 # searcher4 = AStarSearcher(searchProblem.cyclic_simp_delivery_graph) # A*
169 # 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 _____
11 from searchGeneric import AStarSearcher, visualize
12 from searchProblem import Path
13
14 class SearcherMPP(AStarSearcher):
15     """returns a searcher for a problem.
16     Paths can be found by repeatedly calling search().
17     """
18     def __init__(self, problem):
19         super().__init__(problem)
20         self.explored = set()
21
22     @visualize
23     def search(self):
24         """returns next path from an element of problem's start nodes
25         to a goal node.
26         Returns None if no path exists.
27         """
28         while not self.empty_frontier():
29             path = self.frontier.pop()
30             if path.end() not in self.explored:
31                 self.display(2, "Expanding:", path, "(cost:", path.cost, ")")
32                 self.explored.add(path.end())
33                 self.num_expanded += 1
34                 if self.problem.is_goal(path.end()):
35                     self.display(1, self.num_expanded, "paths have been
36                         expanded and",
37                             len(self.frontier), "paths remain in the
38                             frontier")
39                     self.solution = path # store the solution found
40                     return path
41                 else:
42                     neighs = self.problem.neighbors(path.end())
43                     self.display(3, "Neighbors are", neighs)
44                     for arc in neighs:
45                         self.add_to_frontier(Path(path, arc))
46                         self.display(3, "Frontier:", self.frontier)
47             self.display(1, "No (more) solutions. Total of",
48                 self.num_expanded, "paths expanded.")
49
50 from searchGeneric import test
51 if __name__ == "__main__":
52     test(SearcherMPP)
53
54 import searchProblem
55 # searcherMPPcdp = SearcherMPP(searchProblem.cyclic_simp_delivery_graph)
56 # 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
11 from searchProblem import Path
12 from searchGeneric import Searcher
13 from display import Displayable, visualize
14
15 class DF_branch_and_bound(Searcher):
16     """returns a branch and bound searcher for a problem.
17     An optimal path with cost less than bound can be found by calling
18     search()
19     """
20     def __init__(self, problem, bound=float("inf")):
21         """creates a searcher than can be used with search() to find an
22         optimal path.
23         bound gives the initial bound. By default this is infinite -
24         meaning there
25         is no initial pruning due to depth bound
26         """
27         super().__init__(problem)
28         self.best_path = None
29         self.bound = bound
30
31     @visualize
32     def search(self):
33         """returns an optimal solution to a problem with cost less than
34         bound.
35         returns None if there is no solution with cost less than bound."""
36         self.frontier = [Path(self.problem.start_node())]

```

```

33     self.num_expanded = 0
34     while self.frontier:
35         path = self.frontier.pop()
36         if path.cost+self.problem.heuristic(path.end()) < self.bound:
37             # if path.end() not in path.initial_nodes(): # for cycle
38                 pruning
39             self.display(3, "Expanding:", path, "cost:", path.cost)
40             self.num_expanded += 1
41             if self.problem.is_goal(path.end()):
42                 self.best_path = path
43                 self.bound = path.cost
44                 self.display(2, "New best path:", path, " cost:", path.cost)
45             else:
46                 neighs = self.problem.neighbors(path.end())
47                 self.display(3, "Neighbors are", neighs)
48                 for arc in reversed(list(neighs)):
49                     self.add_to_frontier(Path(path, arc))
50             self.display(1, "Number of paths expanded:", self.num_expanded,
51                         "(optimal" if self.best_path else "(no", "solution
52                         found)")
53     self.solution = self.best_path
54     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)
54 from searchGeneric import test
55 if __name__ == "__main__":
56     test(DF_branch_and_bound)
57
58 # Example queries:
59 import searchProblem
60 # searcher1 = DF_branch_and_bound(searchProblem.simp_delivery_graph)
61 # searcher1.search() # find optimal path
62 # searcher2 =
63     DF_branch_and_bound(searchProblem.cyclic_simp_delivery_graph,
64                         bound=100)
65 # searcher2.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
11 from searchGeneric import Searcher, AStarSearcher
12 from searchBranchAndBound import DF_branch_and_bound
13 from searchMPP import SearcherMPP
14
15 DF_branch_and_bound.max_display_level = 1
16 Searcher.max_display_level = 1
17
18 def run(problem,name):
19     print("\n\n*****",name)
20
21     print("\nA*:")
22     asearcher = AStarSearcher(problem)
23     print("Path found:",asearcher.search()," cost=",asearcher.solution.cost)
24     print("there are",asearcher.frontier.count(asearcher.solution.cost),
25           "elements remaining on the queue with
26           f-value=",asearcher.solution.cost)
27
28     print("\nA* with MPP:"),
29     msearcher = SearcherMPP(problem)
30     print("Path found:",msearcher.search()," cost=",msearcher.solution.cost)
31     print("there are",msearcher.frontier.count(msearcher.solution.cost),
32           "elements remaining on the queue with
33           f-value=",msearcher.solution.cost)
34
35     bound = asearcher.solution.cost+0.01
36     print("\nBranch and bound (with too-good initial bound of", bound,")")
37     tbb = DF_branch_and_bound(problem,bound) # cheating!!!!
38     print("Path found:",tbb.search()," cost=",tbb.solution.cost)
39     print("Rerunning B&B")
40     print("Path found:",tbb.search())
41
42     bbound = asearcher.solution.cost*2+10
43     print("\nBranch and bound (with not-very-good initial bound of",
44           bbound, ")")
45     tbb2 = DF_branch_and_bound(problem,bbound) # cheating!!!!
46     print("Path found:",tbb2.search()," cost=",tbb2.solution.cost)
47     print("Rerunning B&B")
48     print("Path found:",tbb2.search())
49
50     print("\nDepth-first search: (Use ^C if it goes on forever)")

```

```
48     tsearcher = Searcher(problem)
49     print("Path found:", tsearcher.search(), " cost=", tsearcher.solution.cost)
50
51
52 import searchProblem
53 from searchTest import run
54 if __name__ == "__main__":
55     run(searchProblem.problem1, "Problem 1")
56     # run(searchProblem.simp_delivery_graph, "Acyclic Delivery")
57     # run(searchProblem.cyclic_simp_delivery_graph, "Cyclic Delivery")
58     # also test some graphs with cycles, and some with multiple least-cost
    paths
```

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 _____
11 import random
12 import matplotlib.pyplot as plt
13
14 class Variable(object):
15     """A random variable.
16     name (string) - name of the variable
17     domain (list) - a list of the values for the variable.
18     Variables are ordered according to their name.
19     """
20
21     def __init__(self, name, domain, position=None):
22         """Variable
23         name a string
24         domain a list of printable values
25         position of form (x,y)
26         """
27         self.name = name # string
28         self.domain = domain # list of values
29         self.position = position if position else (random.random(),
30                                                     random.random())
31         self.size = len(domain)
```

```

32     def __str__(self):
33         return self.name
34
35     def __repr__(self):
36         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_____
11 from variable import Variable
12
13 class Constraint(object):
14     """A Constraint consists of
15     * scope: a tuple of variables
16     * condition: a Boolean function that can applied to a tuple of values
17       for variables in scope
18     * string: a string for printing the constraints. All of the strings
19       must be unique.
20     for the variables
21     """
22     def __init__(self, scope, condition, string=None, position=None):
23         self.scope = scope
24         self.condition = condition
25         if string is None:
26             self.string = f"{self.condition.__name__}({self.scope})"
27         else:
28             self.string = string
29             self.position = position
30
31     def __repr__(self):
32         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)
32 def can_evaluate(self, assignment):
33     """
34     assignment is a variable:value dictionary
35     returns True if the constraint can be evaluated given assignment
36     """
37     return all(v in assignment for v in self.scope)
38
39 def holds(self, assignment):
40     """returns the value of Constraint con evaluated in assignment.
41
42     precondition: all variables are assigned in assignment, ie
43                   self.can_evaluate(assignment) is true
44     """
45     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)
46 class CSP(object):
47     """A CSP consists of
48     * a title (a string)
49     * variables, a set of variables
50     * constraints, a list of constraints
51     * var_to_const, a variable to set of constraints dictionary
52     """
53     def __init__(self, title, variables, constraints):
54         """title is a string
55         variables is set of variables
56         constraints is a list of constraints

```

```

57     """
58     self.title = title
59     self.variables = variables
60     self.constraints = constraints
61     self.var_to_const = {var:set() for var in self.variables}
62     for con in constraints:
63         for var in con.scope:
64             self.var_to_const[var].add(con)
65
66     def __str__(self):
67         """string representation of CSP"""
68         return str(self.title)
69
70     def __repr__(self):
71         """more detailed string representation of CSP"""
72         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)_____
74     def consistent(self,assignment):
75         """assignment is a variable:value dictionary
76         returns True if all of the constraints that can be evaluated
77             evaluate to True given assignment.
78         """
79         return all(con.holds(assignment)
80                     for con in self.constraints
81                     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)_____
83     def show(self):
84         plt.ion() # interactive
85         ax = plt.figure().gca()
86         ax.set_axis_off()
87         plt.title(self.title)
88         var_bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5")
89         con_bbox = dict(boxstyle="square,pad=1.0",color="green")
90         for var in self.variables:
91             if var.position is None:
92                 var.position = (random.random(), random.random())
93         for con in self.constraints:
94             if con.position is None:

```

```

95         con.position = tuple(sum(var.position[i] for var in
96                               con.scope)/len(con.scope)
97                               for i in range(2))
98         bbox = dict(boxstyle="square,pad=1.0",color="green")
99         for var in con.scope:
100             ax.annotate(con.string, var.position, xytext=con.position,
101                         arrowprops={'arrowstyle':'-'},bbox=con_bbox,
102                         ha='center')
103     for var in self.variables:
104         x,y = var.position
105         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_____
11 from cspProblem import Variable, CSP, Constraint
12 from operator import lt,ne,eq,gt
13
14 def ne_(val):
15     """not equal value"""
16     # nev = lambda x: x != val # alternative definition
17     # nev = partial(neq,val) # another alternative definition
18     def nev(x):
19         return val != x
20     nev.__name__ = f"{val} != " # name of the function
21     return nev

```

Similarly *is_*(x)(y) is true when $x = y$.

```

_____cspExamples.py — (continued)_____
23 def is_(val):
24     """is a value"""
25     # isv = lambda x: x == val # alternative definition
26     # isv = partial(eq,val) # another alternative definition
27     def isv(x):
28         return val == x
29     isv.__name__ = f"{val} == "
30     return isv

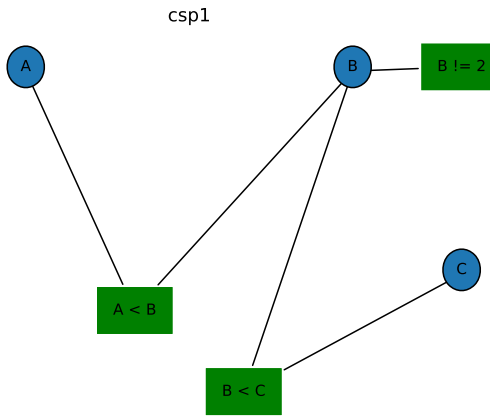
```

The CSP, *csp0* 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()`

```

32 X = Variable('X', {1,2,3})
33 Y = Variable('Y', {1,2,3})
34 Z = Variable('Z', {1,2,3})
35 csp0 = CSP("csp0", {X,Y,Z},
36           [ Constraint([X,Y],lt),
37             Constraint([Y,Z],lt)])

```

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

```

cspExamples.py — (continued)
39 A = Variable('A', {1,2,3,4}, position=(0.2,0.9))
40 B = Variable('B', {1,2,3,4}, position=(0.8,0.9))
41 C = Variable('C', {1,2,3,4}, position=(1,0.4))
42 C0 = Constraint([A,B], lt, "A < B", position=(0.4,0.3))
43 C1 = Constraint([B], ne_(2), "B != 2", position=(1,0.9))
44 C2 = Constraint([B,C], lt, "B < C", position=(0.6,0.1))
45 csp1 = CSP("csp1", {A, B, C},
46           [C0, C1, C2])
47
48 csp1s = CSP("csp1s", {A, B, C},
49            [C0, C2]) # A<B, B<C

```

The next CSP, *csp2* 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

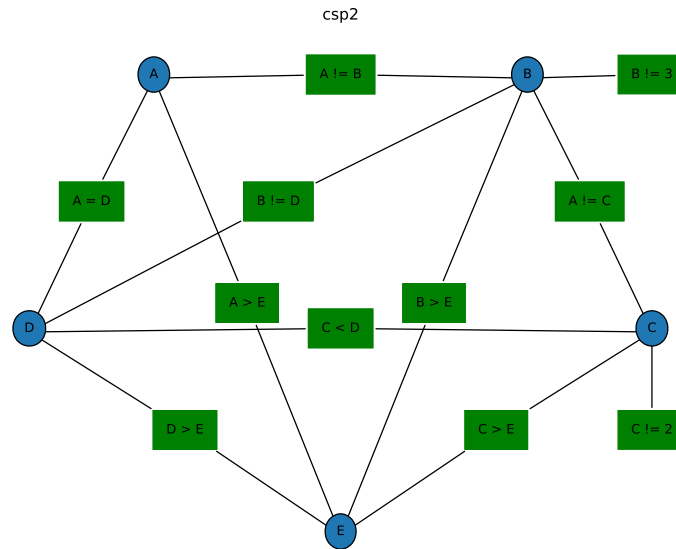


Figure 4.2: csp2.show()

two more.

```

cspExamples.py — (continued)
51 D = Variable('D', {1,2,3,4}, position=(0,0.4))
52 E = Variable('E', {1,2,3,4}, position=(0.5,0))
53 csp2 = CSP("csp2", {A,B,C,D,E},
54     [ Constraint([B], ne_(3), "B != 3", position=(1,0.9)),
55       Constraint([C], ne_(2), "C != 2", position=(1,0.2)),
56       Constraint([A,B], ne, "A != B"),
57       Constraint([B,C], ne, "A != C"),
58       Constraint([C,D], lt, "C < D"),
59       Constraint([A,D], eq, "A = D"),
60       Constraint([E,A], lt, "E < A"),
61       Constraint([E,B], lt, "E < B"),
62       Constraint([E,C], lt, "E < C"),
63       Constraint([E,D], lt, "E < D"),
64       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)
66 csp3 = CSP("csp3", {A,B,C,D,E},
67     [Constraint([A,B], ne, "A != B"),
68     Constraint([A,D], lt, "A < D"),

```

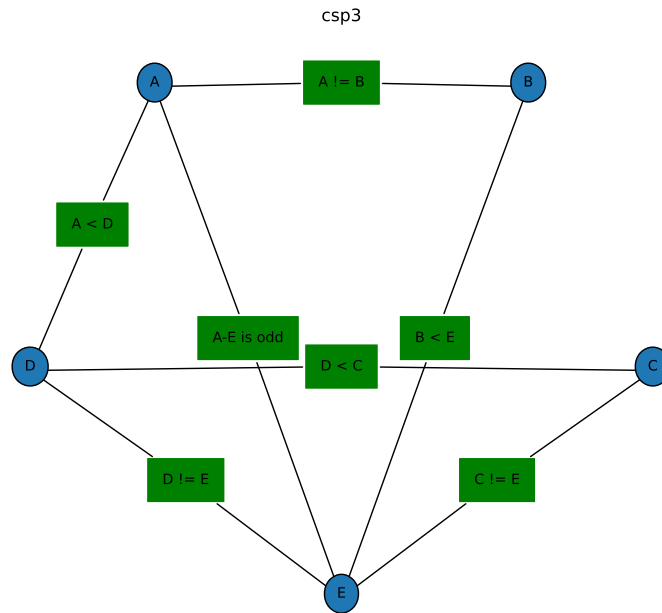


Figure 4.3: csp3.show()

```

69     Constraint([A,E], lambda a,e: (a-e)%2 == 1, "A-E is odd"),
70     Constraint([B,E], lt, "B < E"),
71     Constraint([D,C], lt, "D < C"),
72     Constraint([C,E], ne, "C != E"),
73     Constraint([D,E], ne, "D != E"))

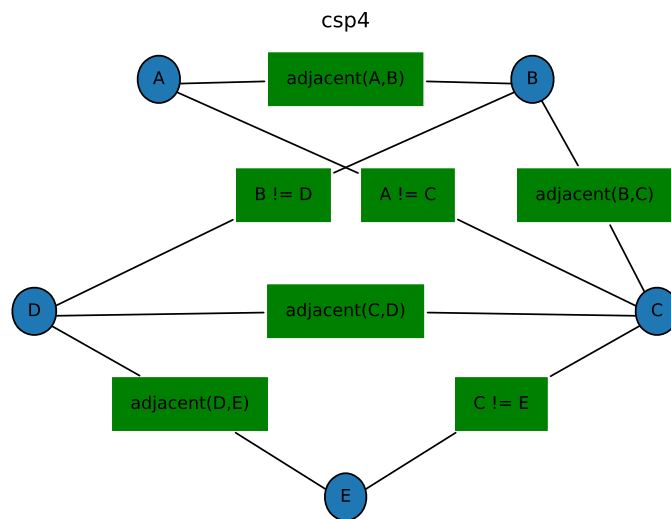
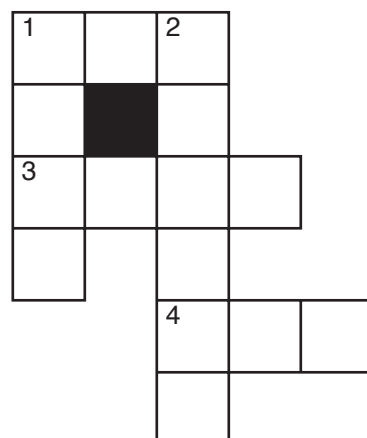
```

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

```

_____cspExamples.py — (continued) _____
75 def adjacent(x,y):
76     """True when x and y are adjacent numbers"""
77     return abs(x-y) == 1
78
79 csp4 = CSP("csp4", {A,B,C,D,E},
80     [Constraint([A,B], adjacent, "adjacent(A,B)"),
81     Constraint([B,C], adjacent, "adjacent(B,C)"),
82     Constraint([C,D], adjacent, "adjacent(C,D)"),
83     Constraint([D,E], adjacent, "adjacent(D,E)"),
84     Constraint([A,C], ne, "A != C"),
85     Constraint([B,D], ne, "B != D"),
86     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

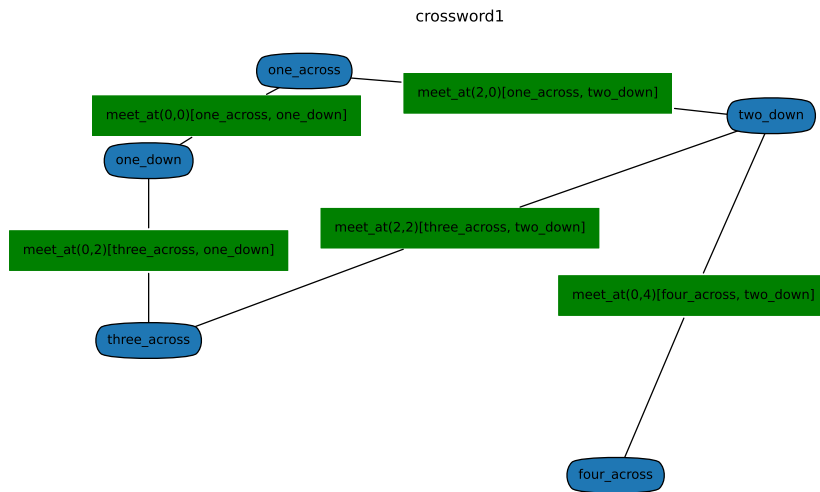


Figure 4.6: crossword1.show()

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)
88 def meet_at(p1,p2):
89     """returns a function of two words that is true
90         when the words intersect at positions p1, p2.
91     The positions are relative to the words; starting at position 0.
92     meet_at(p1,p2)(w1,w2) is true if the same letter is at position p1 of
93         word w1
94         and at position p2 of word w2.
95     """
96     def meets(w1,w2):
97         return w1[p1] == w2[p2]
98     meets.__name__ = f"meet_at({p1},{p2})"
99     return meets
100 one_across = Variable('one_across', {'ant', 'big', 'bus', 'car', 'has'},
101     position=(0.3,0.9))
102 one_down = Variable('one_down', {'book', 'buys', 'hold', 'lane', 'year'},
103     position=(0.1,0.7))
104 two_down = Variable('two_down', {'ginger', 'search', 'symbol', 'syntax'},

```

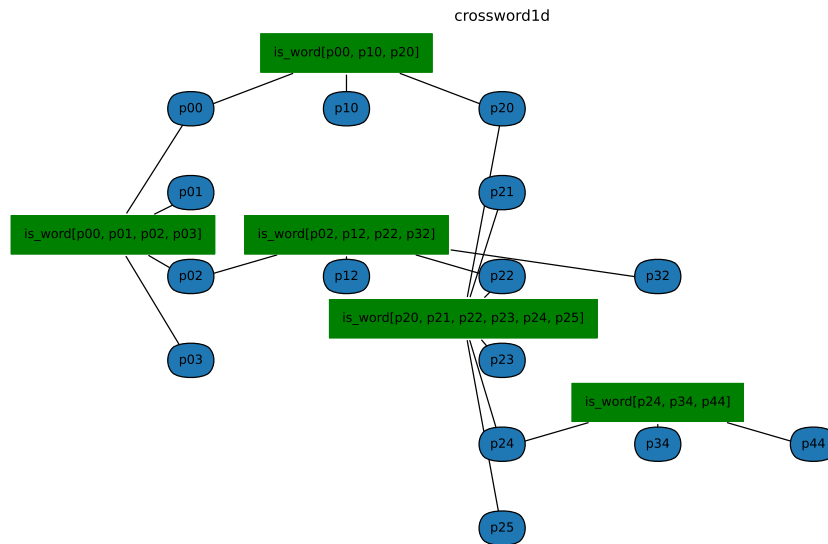



Figure 4.7: crossword1d.show()

```

    position=(0.9,0.8))
103 three_across = Variable('three_across', {'book', 'buys', 'hold', 'land',
    'year'}, position=(0.1,0.3))
104 four_across = Variable('four_across',{'ant', 'big', 'bus', 'car', 'has'},
    position=(0.7,0.0))
105 crossword1 = CSP("crossword1",
106     {one_across, one_down, two_down, three_across,
    four_across},
107     [Constraint([one_across,one_down], meet_at(0,0)),
108     Constraint([one_across,two_down], meet_at(2,0)),
109     Constraint([three_across,two_down], meet_at(2,2)),
110     Constraint([three_across,one_down], meet_at(0,2)),
111     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)
113 words = {'ant', 'big', 'bus', 'car', 'has', 'book', 'buys', 'hold',
114         'lane', 'year', 'ginger', 'search', 'symbol', 'syntax'}
115
116 def is_word(*letters, words=words):

```

```

117     """is true if the letters concatenated form a word in words"""
118     return "".join(letters) in words
119
120 letters = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
121           "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y",
122           "z"}
123
124 # pij is the variable representing the letter i from the left and j down
125     (starting from 0)
126 p00 = Variable('p00', letters, position=(0.1,0.85))
127 p10 = Variable('p10', letters, position=(0.3,0.85))
128 p20 = Variable('p20', letters, position=(0.5,0.85))
129 p01 = Variable('p01', letters, position=(0.1,0.7))
130 p21 = Variable('p21', letters, position=(0.5,0.7))
131 p02 = Variable('p02', letters, position=(0.1,0.55))
132 p12 = Variable('p12', letters, position=(0.3,0.55))
133 p22 = Variable('p22', letters, position=(0.5,0.55))
134 p32 = Variable('p32', letters, position=(0.7,0.55))
135 p03 = Variable('p03', letters, position=(0.1,0.4))
136 p23 = Variable('p23', letters, position=(0.5,0.4))
137 p24 = Variable('p24', letters, position=(0.5,0.25))
138 p34 = Variable('p34', letters, position=(0.7,0.25))
139 p44 = Variable('p44', letters, position=(0.9,0.25))
140 p25 = Variable('p25', letters, position=(0.5,0.1))
141
142 crossword1d = CSP("crossword1d",
143                  {p00, p10, p20, # first row
144                    p01, p21, # second row
145                    p02, p12, p22, p32, # third row
146                    p03, p23, #fourth row
147                    p24, p34, p44, # fifth row
148                    p25 # sixth row
149                  },
150                  [Constraint([p00, p10, p20], is_word,
151                             position=(0.3,0.95)), # 1-across
152                    Constraint([p00, p01, p02, p03], is_word,
153                             position=(0,0.625)), # 1-down
154                    Constraint([p02, p12, p22, p32], is_word,
155                             position=(0.3,0.625)), # 3-across
156                    Constraint([p20, p21, p22, p23, p24, p25], is_word,
157                             position=(0.45,0.475)), # 2-down
158                    Constraint([p24, p34, p44], is_word,
159                             position=(0.7,0.325)) # 4-across
160                  ])

```

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)_____
156 def queens(ri,rj):
157     """ri and rj are different rows, return the condition that the queens
        cannot take each other"""
158     def no_take(ci,cj):
159         """is true if queen at (ri,ci) cannot take a queen at (rj,cj)"""
160         return ci != cj and abs(ri-ci) != abs(rj-cj)
161     return no_take
162
163 def n_queens(n):
164     """returns a CSP for n-queens"""
165     columns = list(range(n))
166     variables = [Variable(f"R{i}",columns) for i in range(n)]
167     return CSP("n-queens",
168               variables,
169               [Constraint([variables[i], variables[j]], queens(i,j))
170                  for i in range(n) for j in range(n) if i != j])
171
172 # try the CSP n_queens(8) in one of the solvers.
173 # 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)_____
175 def test_csp(CSP_solver, csp=csp1,
176             solutions=[{A: 1, B: 3, C: 4}, {A: 2, B: 3, C: 4}]):
177     """CSP_solver is a solver that takes a csp and returns a solution
178     csp is a constraint satisfaction problem
179     solutions is the list of all solutions to csp
180     This tests whether the solution returned by CSP_solver is a solution.
181     """
182     print("Testing csp with",CSP_solver.__doc__)
183     sol0 = CSP_solver(csp)
184     print("Solution found:",sol0)
185     assert sol0 in solutions, f"Solution not correct for {csp}"
186     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).

```

11 from cspExamples import csp1,csp1s,csp2,test_csp, crossword1, crossword1d
12
13 def dfs_solver(constraints, context, var_order):
14     """generator for all solutions to csp.
15     context is an assignment of values to some of the variables.
16     var_order is a list of the variables in csp that are not in context.
17     """
18     to_eval = {c for c in constraints if c.can_evaluate(context)}
19     if all(c.holds(context) for c in to_eval):
20         if var_order == []:
21             yield context
22         else:
23             rem_cons = [c for c in constraints if c not in to_eval]
24             var = var_order[0]
25             for val in var.domain:
26                 yield from dfs_solver(rem_cons, context|{var:val},
27                                     var_order[1:])
28
29 def dfs_solve_all(csp, var_order=None):
30     """depth-first CSP solver to return a list of all solutions to csp.
31     """
32     if var_order == None: # use an arbitrary variable order
33         var_order = list(csp.variables)
34     return list( dfs_solver(csp.constraints, {}, var_order))
35
36 def dfs_solve1(csp, var_order=None):
37     """depth-first CSP solver to find single solution or None if there are
38     no solutions.
39     """
40     if var_order == None: # use an arbitrary variable order
41         var_order = list(csp.variables)

```

```

40     gen = dfs_solver(csp.constraints, {}, var_order)
41     try:      # Python generators raise an exception if there are no more
               elements.
42         return next(gen)
43     except StopIteration:
44         return None
45
46 if __name__ == "__main__":
47     test_csp(dfs_solve1)
48
49 #Try:
50 # dfs_solve_all(csp1)
51 # dfs_solve_all(csp2)
52 # dfs_solve_all(crossword1)
53 # 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 its variables are assigned. Given an elimination ordering, it is possible to determine when each constraint needs to be tested. Implement this. Hint: create a parallel list of sets of constraints, where at each position i in the list, the constraints at position i can be evaluated when the variable at position i has been assigned.

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

4.3 Converting CSPs to Search Problems

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

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

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

```

cspSearch.py — Representations of a Search Problem from a CSP.
11 from cspProblem import CSP, Constraint
12 from searchProblem import Arc, Search_problem
13
14 class Search_from_CSP(Search_problem):
15     """A search problem directly from the CSP.
16
17     A node is a variable:value dictionary"""
18     def __init__(self, csp, variable_order=None):
19         self.csp=csp
20         if variable_order:
21             assert set(variable_order) == set(csp.variables)
22             assert len(variable_order) == len(csp.variables)
23             self.variables = variable_order
24         else:
25             self.variables = list(csp.variables)
26
27     def is_goal(self, node):
28         """returns whether the current node is a goal for the search
29         """
30         return len(node)==len(self.csp.variables)
31
32     def start_node(self):
33         """returns the start node for the search
34         """
35         return {}

```

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

```

cspSearch.py — (continued)
37 def neighbors(self, node):
38     """returns a list of the neighboring nodes of node.
39     """
40     var = self.variables[len(node)] # the next variable
41     res = []
42     for val in var.domain:
43         new_env = node|{var:val} #dictionary union
44         if self.csp.consistent(new_env):
45             res.append(Arc(node,new_env))
46     return res

```

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

```

cspSearch.py — (continued)
48 from cspExamples import csp1,csp1a,csp2,test_csp, crossword1, crossword1d
49 from searchGeneric import Searcher

```

```

50
51 def solver_from_searcher(csp):
52     """depth-first search solver"""
53     path = Searcher(Search_from_CSP(csp)).search()
54     if path is not None:
55         return path.end()
56     else:
57         return None
58
59 if __name__ == "__main__":
60     test_csp(solver_from_searcher)
61
62 ## Test Solving CSPs with Search:
63 searcher1 = Searcher(Search_from_CSP(csp1))
64 #print(searcher1.search()) # get next solution
65 searcher2 = Searcher(Search_from_CSP(csp2))
66 #print(searcher2.search()) # get next solution
67 searcher3 = Searcher(Search_from_CSP(crossword1))
68 #print(searcher3.search()) # get next solution
69 searcher4 = Searcher(Search_from_CSP(crossword1d))
70 #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_____
11 from display import Displayable
12
13 class Con_solver(Displayable):
14     """Solves a CSP with arc consistency and domain splitting
15     """
16     def __init__(self, csp, **kwargs):
17         """a CSP solver that uses arc consistency
18         * csp is the CSP to be solved
19         * kwargs is the keyword arguments for Displayable superclass

```

```

20     """
21     self.csp = csp
22     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)_____
24 def make_arc_consistent(self, orig_domains=None, to_do=None):
25     """Makes this CSP arc-consistent using generalized arc consistency
26     orig_domains is the original domains
27     to_do is a set of (variable,constraint) pairs
28     returns the reduced domains (an arc-consistent variable:domain
        dictionary)
29     """
30     if orig_domains is None:
31         orig_domains = {var:var.domain for var in self.csp.variables}
32     if to_do is None:
33         to_do = {(var, const) for const in self.csp.constraints
34                 for var in const.scope}
35     else:
36         to_do = to_do.copy() # use a copy of to_do
37     domains = orig_domains.copy()
38     self.display(2,"Performing AC with domains", domains)
39     while to_do:
40         var, const = self.select_arc(to_do)
41         self.display(3, "Processing arc (" , var , "," , const , ")")
42         other_vars = [ov for ov in const.scope if ov != var]
43         new_domain = {val for val in domains[var]
44                       if self.any_holds(domains, const, {var: val},
45                                         other_vars)}
46         if new_domain != domains[var]:
47             self.display(4, "Arc: (" , var , "," , const , ") is
                inconsistent")
48             self.display(3, "Domain pruned", "dom(" , var , ") =",
                new_domain,
49                 " due to " , const)
50             domains[var] = new_domain
51             add_to_do = self.new_to_do(var, const) - to_do
52             to_do |= add_to_do # set union
53             self.display(3, " adding", add_to_do if add_to_do else
                "nothing", "to to_do.")
54             self.display(4, "Arc: (" , var , "," , const , ") now consistent")
55             self.display(2, "AC done. Reduced domains", domains)
56         return domains
57
58 def new_to_do(self, var, const):
59     """returns new elements to be added to to_do after assigning
        variable var in constraint const.

```



```

60 |         """
61 |         return {(nvar, nconst) for nconst in self.csp.var_to_const[var]
62 |                if nconst != const
63 |                for nvar in nconst.scope
64 |                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)
66 | def select_arc(self, to_do):
67 |     """Selects the arc to be taken from to_do .
68 |     * to_do is a set of arcs, where an arc is a (variable,constraint)
69 |       pair
70 |     the element selected must be removed from to_do.
71 |     """
    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 find 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)
73 | def any_holds(self, domains, const, env, other_vars, ind=0):
74 |     """returns True if Constraint const holds for an assignment

```

```

75     that extends env with the variables in other_vars[ind:]
76     env is a dictionary
77     """
78     if ind == len(other_vars):
79         return const.holds(env)
80     else:
81         var = other_vars[ind]
82         for val in domains[var]:
83             if self.any_holds(domains, const, env|{var:val}, other_vars,
84                             ind + 1):
85                 return True
86         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)_____
87 def solve_one(self, domains=None, to_do=None):
88     """return a solution to the current CSP or False if there are no
89     solutions
90     to_do is the list of arcs to check
91     """
92     new_domains = self.make_arc_consistent(domains, to_do)
93     if any(len(new_domains[var]) == 0 for var in new_domains):
94         return False
95     elif all(len(new_domains[var]) == 1 for var in new_domains):
96         self.display(2, "solution:", {var: select(
97             new_domains[var] for var in new_domains)})
98         return {var: select(new_domains[var] for var in new_domains)}
99     else:
100         var = self.select_var(x for x in self.csp.variables if
101                               len(new_domains[x]) > 1)
102         if var:
103             dom1, dom2 = partition_domain(new_domains[var])
104             self.display(3, "...splitting", var, "into", dom1, "and",
105                         dom2)
106             new_doms1 = copy_with_assign(new_domains, var, dom1)
107             new_doms2 = copy_with_assign(new_domains, var, dom2)
108             to_do = self.new_to_do(var, None)
109             self.display(3, " adding", to_do if to_do else "nothing",
110                         "to to_do.")
111             return self.solve_one(new_doms1, to_do) or
112                    self.solve_one(new_doms2, to_do)
113
114 def select_var(self, iter_vars):
115     """return the next variable to split"""
116     return select(iter_vars)

```

```

112
113 def partition_domain(dom):
114     """partitions domain dom into two.
115     """
116     split = len(dom) // 2
117     dom1 = set(list(dom)[:split])
118     dom2 = dom - dom1
119     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)_____
121 def copy_with_assign(domains, var=None, new_domain={True, False}):
122     """create a copy of the domains with an assignment var=new_domain
123     if var==None then it is just a copy.
124     """
125     newdoms = domains.copy()
126     if var is not None:
127         newdoms[var] = new_domain
128     return newdoms

```

```

_____cspConsistency.py — (continued)_____
130 def select(iterable):
131     """select an element of iterable. Returns None if there is no such
132     element.
133
134     This implementation just picks the first element.
135     For many of the uses, which element is selected does not affect
136     correctness,
137     but may affect efficiency.
138     """
139     for e in iterable:
140         return e # returns first element found

```

Exercise 4.10 Implement *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)_____
140 from cspExamples import test_csp
141 def ac_solver(csp):
142     "arc consistency (solve_one)"

```

```

143     return Con_solver(csp).solve_one()
144
145 if __name__ == "__main__":
146     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)_____
148 from searchProblem import Arc, Search_problem
149
150 class Search_with_AC_from_CSP(Search_problem, Displayable):
151     """A search problem with arc consistency and domain splitting
152
153     A node is a CSP """
154     def __init__(self, csp):
155         self.cons = Con_solver(csp) #copy of the CSP
156         self.domains = self.cons.make_arc_consistent()
157
158     def is_goal(self, node):
159         """node is a goal if all domains have 1 element"""
160         return all(len(node[var])==1 for var in node)
161
162     def start_node(self):
163         return self.domains
164
165     def neighbors(self, node):
166         """returns the neighboring nodes of node.
167         """
168         neighs = []
169         var = select(x for x in node if len(node[x])>1)
170         if var:
171             dom1, dom2 = partition_domain(node[var])
172             self.display(2, "Splitting", var, "into", dom1, "and", dom2)
173             to_do = self.cons.new_to_do(var, None)
174             for dom in [dom1, dom2]:
175                 newdoms = copy_with_assign(node, var, dom)
176                 cons_doms = self.cons.make_arc_consistent(newdoms, to_do)
177                 if all(len(cons_doms[v])>0 for v in cons_doms):
178                     # all domains are non-empty
179                     neighs.append(Arc(node, cons_doms))
180             else:
181                 self.display(2, "...", var, "in", dom, "has no solution")
182         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) _____
184 from cspExamples import test_csp
185 from searchGeneric import Searcher
186
187 def ac_search_solver(csp):
188     """arc consistency (search interface)"""
189     sol = Searcher(Search_with_AC_from_CSP(csp)).search()
190     if sol:
191         return {v:select(d) for (v,d) in sol.end().items()}
192
193 if __name__ == "__main__":
194     test_csp(ac_search_solver)

```

Testing:

```

_____cspConsistency.py — (continued) _____
196 from cspExamples import csp1, csp1s, csp2, csp3, csp4, crossword1,
    crossword1d
197
198 ## Test Solving CSPs with Arc consistency and domain splitting:
199 #Con_solver.max_display_level = 4 # display details of AC (0 turns off)
200 #Con_solver(csp1).solve_one()
201 #searcher1d = Searcher(Search_with_AC_from_CSP(csp1))
202 #print(searcher1d.search())
203 #Searcher.max_display_level = 2 # display search trace (0 turns off)
204 #searcher2c = Searcher(Search_with_AC_from_CSP(csp2))
205 #print(searcher2c.search())
206 #searcher3c = Searcher(Search_with_AC_from_CSP(crossword1))
207 #print(searcher3c.search())
208 #searcher4c = Searcher(Search_with_AC_from_CSP(crossword1d))
209 #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
11 from cspProblem import CSP, Constraint
12 from searchProblem import Arc, Search_problem
13 from display import Displayable
14 import random
15 import heapq
16
17 class SLSearcher(Displayable):
18     """A search problem directly from the CSP..
19
20     A node is a variable:value dictionary"""
21     def __init__(self, csp):
22         self.csp = csp
23         self.variables_to_select = {var for var in self.csp.variables
24                                   if len(var.domain) > 1}
25         # Create assignment and conflicts set
26         self.current_assignment = None # this will trigger a random restart
27         self.number_of_steps = 0 #number of steps after the initialization

```

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

```

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

```

The *search* method is the top-level searching algorithm. It can either be used to start the search or to continue searching. If there is no current assignment, it must create one. Note that, when counting steps, a restart is counted as one

step, which is not appropriate for CSPs with many variables, as it is a relatively expensive operation for these cases.

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

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

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

```

cspSLS.py — (continued)
42 def search(self, max_steps, prob_best=0, prob_anycon=1.0):
43     """
44     returns the number of steps or None if there is no solution.
45     If there is a solution, it can be found in self.current_assignment
46
47     max_steps is the maximum number of steps it will try before giving
48     up
49     prob_best is the probability that a best variable (one in most
50     conflict) is selected
51     prob_anycon is the probability that a variable in any conflict is
52     selected
53     (otherwise a variable is chosen at random)
54     """
55     if self.current_assignment is None:
56         self.restart()
57         self.number_of_steps += 1
58         if not self.conflicts:
59             self.display(1, "Solution found:", self.current_assignment,
60                         "after restart")
61             return self.number_of_steps
62     if prob_best > 0: # we need to maintain a variable priority queue
63         return self.search_with_var_pq(max_steps, prob_best,
64                                         prob_anycon)
65     else:
66         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)
63 def search_with_any_conflict(self, max_steps, probab_anycon=1.0):
64     """Searches with the any_conflict heuristic.
65     This relies on just maintaining the set of conflicts;
66     it does not maintain a priority queue
67     """
68     self.variable_pq = None # we are not maintaining the priority queue.
69                             # This ensures it is regenerated if
70                             # we call search_with_var_pq.
71     for i in range(max_steps):
72         self.number_of_steps +=1
73         if random.random() < probab_anycon:
74             con = random.choice(self.conflicts) # pick random conflict
75             var = random.choice(con.scope) # pick variable in conflict
76         else:
77             var = random.choice(self.variables_to_select)
78         if len(var.domain) > 1:
79             val = random.choice([val for val in var.domain
80                                if val is not
81                                self.current_assignment[var]])
82             self.display(2,self.number_of_steps,":
83             Assigning",var,"=",val)
84             self.current_assignment[var]=val
85             for varcon in self.csp.var_to_const[var]:
86                 if varcon.holds(self.current_assignment):
87                     if varcon in self.conflicts:
88                         self.conflicts.remove(varcon)
89                     else:
90                         if varcon not in self.conflicts:
91                             self.conflicts.add(varcon)
92             self.display(2,"  Number of conflicts",len(self.conflicts))
93         if not self.conflicts:
94             self.display(1,"Solution found:", self.current_assignment,
95                         "in", self.number_of_steps,"steps")
96             return self.number_of_steps
97         self.display(1,"No solution in",self.number_of_steps,"steps",
98                     len(self.conflicts),"conflicts remain")
99     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 its 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)
99  def search_with_var_pq(self,max_steps, prob_best=1.0, prob_anycon=1.0):
100      """search with a priority queue of variables.
101      This is used to select a variable with the most conflicts.
102      """
103      if not self.variable_pq:
104          self.create_pq()
105      pick_best_or_con = prob_best + prob_anycon
106      for i in range(max_steps):
107          self.number_of_steps +=1
108          randnum = random.random()
109          ## Pick a variable
110          if randnum < prob_best: # pick best variable
111              var,oldval = self.variable_pq.top()
112          elif randnum < pick_best_or_con: # pick a variable in a conflict
113              con = random_choice(self.conflicts)
114              var = random_choice(con.scope)
115          else: #pick any variable that can be selected
116              var = random_choice(self.variables_to_select)
117          if len(var.domain) > 1: # var has other values
118              ## Pick a value
119              val = random_choice([val for val in var.domain if val is not
120                                self.current_assignment[var]])
121              self.display(2,"Assigning",var,val)
122              ## Update the priority queue
123              var_differential = {}
124              self.current_assignment[var]=val
125              for varcon in self.csp.var_to_const[var]:
126                  self.display(3,"Checking",varcon)
127                  if varcon.holds(self.current_assignment):
128                      if varcon in self.conflicts: #was incons, now consis
129                          self.display(3,"Became consistent",varcon)
130                          self.conflicts.remove(varcon)

```

```

131         for v in varcon.scope: # v is in one fewer
132             conflicts
133             var_differential[v] =
134                 var_differential.get(v,0)-1
135         else:
136             if varcon not in self.conflicts: # was consis, not now
137                 self.display(3,"Became inconsistent",varcon)
138                 self.conflicts.add(varcon)
139                 for v in varcon.scope: # v is in one more
140                     conflicts
141                     var_differential[v] =
142                         var_differential.get(v,0)+1
143                 self.variable_pq.update_each_priority(var_differential)
144                 self.display(2,"Number of conflicts",len(self.conflicts))
145             if not self.conflicts: # no conflicts, so solution found
146                 self.display(1,"Solution found:",
147                     self.current_assignment,"in",
148                     self.number_of_steps,"steps")
149                 return self.number_of_steps
150             self.display(1,"No solution in",self.number_of_steps,"steps",
151                 len(self.conflicts),"conflicts remain")
152         return None

```

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

cspSLS.py — (continued)

```

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

```

cspSLS.py — (continued)

```

165 def random_choice(st):
166     """selects a random element from set st.

```

```

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

```

Exercise 4.15 This makes no attempt to find the best alternative value for a variable. Modify the code so that after selecting a variable it selects a value that reduces the number of conflicts by the most. Have a parameter that specifies the probability that the best value is chosen.

Exercise 4.16 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)
171 | class Updatable_priority_queue(object):
172 |     """A priority queue where the values can be updated.
173 |     Elements with the same value are ordered randomly.
174 |
175 |     This code is based on the ideas described in
176 |     http://docs.python.org/3.3/library/heapq.html
177 |     It could probably be done more efficiently by
178 |     shuffling the modified element in the heap.
179 |     """
180 |     def __init__(self):
181 |         self.pq = [] # priority queue of [val,rand,elt] triples
182 |         self.elt_map = {} # map from elt to [val,rand,elt] triple in pq
183 |         self.REMOVED = "*removed*" # a string that won't be a legal element
184 |         self.max_size=0
185 |
186 |     def add(self,elt,val):
187 |         """adds elt to the priority queue with priority=val.
188 |         """
189 |         assert val <= 0,val

```

```

190     assert elt not in self.elt_map, elt
191     new_triple = [val, random.random(),elt]
192     heapq.heappush(self.pq, new_triple)
193     self.elt_map[elt] = new_triple
194
195     def remove(self,elt):
196         """remove the element from the priority queue"""
197         if elt in self.elt_map:
198             self.elt_map[elt][2] = self.REMOVED
199             del self.elt_map[elt]
200
201     def update_each_priority(self,update_dict):
202         """update values in the priority queue by subtracting the values in
203         update_dict from the priority of those elements in priority queue.
204         """
205         for elt,incr in update_dict.items():
206             if incr != 0:
207                 newval = self.elt_map.get(elt,[0])[0] - incr
208                 assert newval <= 0, f"{elt}:{newval+incr}-{incr}"
209                 self.remove(elt)
210                 if newval != 0:
211                     self.add(elt,newval)
212
213     def pop(self):
214         """Removes and returns the (elt,value) pair with minimal value.
215         If the priority queue is empty, IndexError is raised.
216         """
217         self.max_size = max(self.max_size, len(self.pq)) # keep statistics
218         triple = heapq.heappop(self.pq)
219         while triple[2] == self.REMOVED:
220             triple = heapq.heappop(self.pq)
221         del self.elt_map[triple[2]]
222         return triple[2], triple[0] # elt, value
223
224     def top(self):
225         """Returns the (elt,value) pair with minimal value, without
226         removing it.
227         If the priority queue is empty, IndexError is raised.
228         """
229         self.max_size = max(self.max_size, len(self.pq)) # keep statistics
230         triple = self.pq[0]
231         while triple[2] == self.REMOVED:
232             heapq.heappop(self.pq)
233             triple = self.pq[0]
234         return triple[2], triple[0] # elt, value
235
236     def empty(self):
237         """returns True iff the priority queue is empty"""
238         return all(triple[2] == self.REMOVED for triple in self.pq)

```

4.5.4 Plotting Run-Time Distributions

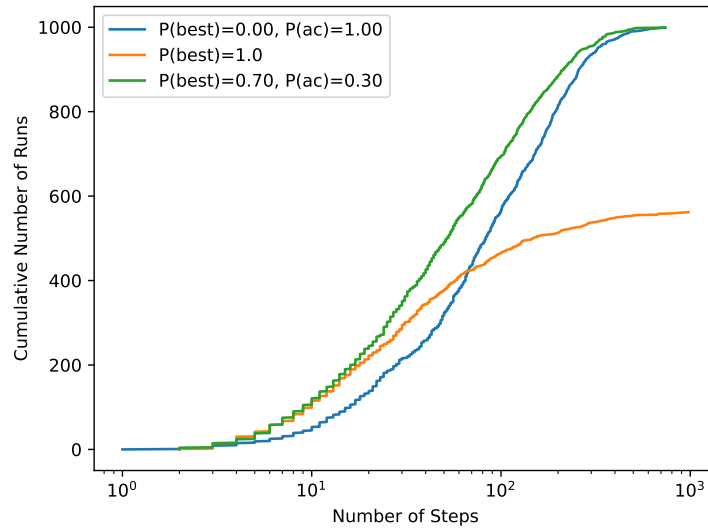
Runtime_distribution uses matplotlib to plot run time distributions. Here the run time is a misnomer as we are only plotting the number of steps, not the time. Computing the run time is non-trivial as many of the runs have a very short run time. To compute the time accurately would require running the 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)
239 import matplotlib.pyplot as plt
240 # plt.style.use('grayscale')
241
242 class Runtime_distribution(object):
243     def __init__(self, csp, xscale='log'):
244         """Sets up plotting for csp
245         xscale is either 'linear' or 'log'
246         """
247         self.csp = csp
248         plt.ion()
249         plt.xlabel("Number of Steps")
250         plt.ylabel("Cumulative Number of Runs")
251         plt.xscale(xscale) # Makes a 'log' or 'linear' scale
252
253     def plot_runs(self, num_runs=100, max_steps=1000, prob_best=1.0,
254                  prob_anycon=1.0):
255         """Plots num_runs of SLS for the given settings.
256         """
257         stats = []
258         SLSearcher.max_display_level, temp_mdl = 0,
259         SLSearcher.max_display_level # no display
260         for i in range(num_runs):
261             searcher = SLSearcher(self.csp)
262             num_steps = searcher.search(max_steps, prob_best, prob_anycon)
263             if num_steps:
264                 stats.append(num_steps)
265             stats.sort()
266             if prob_best >= 1.0:
267                 label = "P(best)=1.0"
268             else:
269                 p_ac = min(prob_anycon, 1-prob_best)
270                 label = "P(best)=%.2f, P(ac)=%.2f" % (prob_best, p_ac)
271             plt.plot(stats, range(len(stats)), label=label)
272             plt.legend(loc="upper left")
273             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 *csp2*.

4.5.5 Testing

```

cspSLS.py — (continued)
273 from cspExamples import test_csp
274 def sls_solver(csp,prob_best=0.7):
275     """stochastic local searcher (prob_best=0.7)"""
276     se0 = SLSearcher(csp)
277     se0.search(1000,prob_best)
278     return se0.current_assignment
279 def any_conflict_solver(csp):
280     """stochastic local searcher (any-conflict)"""
281     return sls_solver(csp,0)
282
283 if __name__ == "__main__":
284     test_csp(sls_solver)
285     test_csp(any_conflict_solver)
286
287 from cspExamples import csp1, csp1s, csp2, crossword1, crossword1d
288
289 ## Test Solving CSPs with Search:
290 #se1 = SLSearcher(csp1); print(se1.search(100))
291 #se2 = SLSearcher(csp2); print(se2.search(1000,1.0)) # greedy
292 #se2 = SLSearcher(csp2); print(se2.search(1000,0)) # any_conflict
293 #se2 = SLSearcher(csp2); print(se2.search(1000,0.7)) # 70% greedy; 30%
    any_conflict
294 #SLSearcher.max_display_level=2 #more detailed display

```

```

295 #se3 = SLSearcher(crossword1); print(se3.search(100),0.7)
296 #p = Runtime_distribution(csp2)
297 #p.plot_runs(1000,1000,0) # any_conflict
298 #p.plot_runs(1000,1000,1.0) # greedy
299 #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_____
11 from cspProblem import Variable, Constraint, CSP
12 class SoftConstraint(Constraint):
13     """A Constraint consists of
14     * scope: a tuple of variables
15     * function: a real-valued function that can applied to a tuple of values
16     * string: a string for printing the constraints. All of the strings
17       must be unique.
18     for the variables
19     """
20     def __init__(self, scope, function, string=None, position=None):
21         Constraint.__init__(self, scope, function, string, position)
22
23     def value(self, assignment):
24         return self.holds(assignment)

```

```

_____cspSoft.py — (continued)_____
25 A = Variable('A', {1,2}, position=(0.2,0.9))
26 B = Variable('B', {1,2,3}, position=(0.8,0.9))
27 C = Variable('C', {1,2}, position=(0.5,0.5))
28 D = Variable('D', {1,2}, position=(0.8,0.1))
29
30 def c1fun(a,b):
31     if a==1: return (5 if b==1 else 2)
32     else: return (0 if b==1 else 4 if b==2 else 3)

```

```

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

```

cspSoft.py — (continued)
60 from display import Displayable, visualize
61 import math
62
63 class DF_branch_and_bound_opt(Displayable):
64     """returns a branch and bound searcher for a problem.
65     An optimal assignment with cost less than bound can be found by calling
66     search()
67     """
68     def __init__(self, csp, bound=math.inf):
69         """creates a searcher than can be used with search() to find an
70         optimal path.
71         bound gives the initial bound. By default this is infinite -
72         meaning there
73         is no initial pruning due to depth bound
74         """
75         super().__init__()

```



```

73         self.csp = csp
74         self.best_asst = None
75         self.bound = bound
76
77     def optimize(self):
78         """returns an optimal solution to a problem with cost less than
79         bound.
80         returns None if there is no solution with cost less than bound."""
81         self.num_expanded=0
82         self.cbsearch({}, 0, self.csp.constraints)
83         self.display(1,"Number of paths expanded:",self.num_expanded)
84         return self.best_asst, self.bound
85
86     def cbsearch(self, asst, cost, constraints):
87         """finds the optimal solution that extends path and is less the
88         bound"""
89         self.display(2,"cbsearch:",asst,cost,constraints)
90         can_eval = [c for c in constraints if c.can_evaluate(asst)]
91         rem_cons = [c for c in constraints if c not in can_eval]
92         newcost = cost + sum(c.value(asst) for c in can_eval)
93         self.display(2,"Evaluating:",can_eval,"cost:",newcost)
94         if newcost < self.bound:
95             self.num_expanded += 1
96             if rem_cons==[]:
97                 self.best_asst = asst
98                 self.bound = newcost
99                 self.display(1,"New best assignment:",asst," cost:",newcost)
100             else:
101                 var = next(var for var in self.csp.variables if var not in
102                             asst)
103                 for val in var.domain:
104                     self.cbsearch({var:val}|asst, newcost, rem_cons)
105
106 # bnb = DF_branch_and_bound_opt(scsp1)
107 # bnb.max_display_level=3 # show more detail
108 # 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.

Propositions and Inference

5.1 Representing Knowledge Bases

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

```
_____logicProblem.py — Representations Logics _____
11 class Clause(object):
12     """A definite clause"""
13
14     def __init__(self, head, body=[]):
15         """clause with atom head and list of atoms body"""
16         self.head = head
17         self.body = body
18
19     def __repr__(self):
20         """returns the string representation of a clause.
21         """
22         if self.body:
23             return self.head + " <- " + " & ".join(str(a) for a in
24                 self.body) + "\n"
25         else:
26             return self.head + "."
```

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

```
_____logicProblem.py — (continued) _____
27 class Askable(object):
28     """An askable atom"""
29
30     def __init__(self, atom):
```

```

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

```

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

logicProblem.py — (continued)

```

42 from display import Displayable
43
44 class KB(Displayable):
45     """A knowledge base consists of a set of clauses.
46     This also creates a dictionary to give fast access to the clauses with
47     an atom in head.
48     """
49     def __init__(self, statements=[]):
50         self.statements = statements
51         self.clauses = [c for c in statements if isinstance(c, Clause)]
52         self.askables = [c.atom for c in statements if isinstance(c,
53             Askable)]
54         self.atom_to_clauses = {} # dictionary giving clauses with atom as
55             head
56         for c in self.clauses:
57             self.add_clause(c)
58
59     def add_clause(self, c):
60         if c.head in self.atom_to_clauses:
61             self.atom_to_clauses[c.head].add(c)
62         else:
63             self.atom_to_clauses[c.head] = {c}
64
65     def clauses_for_atom(self, a):
66         """returns set of clauses with atom a as the head"""
67         if a in self.atom_to_clauses:
68             return self.atom_to_clauses[a]
69         else:
70             return set()
71
72     def __str__(self):
73         """returns a string representation of this knowledge base.
74         """
75         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)
74 triv_KB = KB([
75     Clause('i_am', ['i_think']),
76     Clause('i_think'),
77     Clause('i_smell', ['i_exist'])
78 ])

```

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

```

logicProblem.py — (continued)
80 elect = KB([
81     Clause('light_l1'),
82     Clause('light_l2'),
83     Clause('ok_l1'),
84     Clause('ok_l2'),
85     Clause('ok_cb1'),
86     Clause('ok_cb2'),
87     Clause('live_outside'),
88     Clause('live_l1', ['live_w0']),
89     Clause('live_w0', ['up_s2', 'live_w1']),
90     Clause('live_w0', ['down_s2', 'live_w2']),
91     Clause('live_w1', ['up_s1', 'live_w3']),
92     Clause('live_w2', ['down_s1', 'live_w3']),
93     Clause('live_l2', ['live_w4']),
94     Clause('live_w4', ['up_s3', 'live_w3']),
95     Clause('live_p1', ['live_w3']),
96     Clause('live_w3', ['live_w5', 'ok_cb1']),
97     Clause('live_p2', ['live_w6']),
98     Clause('live_w6', ['live_w5', 'ok_cb2']),
99     Clause('live_w5', ['live_outside']),
100    Clause('lit_l1', ['light_l1', 'live_l1', 'ok_l1']),
101    Clause('lit_l2', ['light_l2', 'live_l2', 'ok_l2']),
102    Askable('up_s1'),
103    Askable('down_s1'),
104    Askable('up_s2'),
105    Askable('down_s2'),
106    Askable('up_s3'),
107    Askable('down_s2')
108 ])
109
110 # 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)
111 elect_bug = KB([
112     Clause('light_l2'),
113     Clause('ok_l1'),
114     Clause('ok_l2'),

```

```

115     Clause('ok_cb1'),
116     Clause('ok_cb2'),
117     Clause('live_outside'),
118     Clause('live_p_2', ['live_w6']),
119     Clause('live_w6', ['live_w5', 'ok_cb2']),
120     Clause('light_l1'),
121     Clause('live_w5', ['live_outside']),
122     Clause('lit_l1', ['light_l1', 'live_l1', 'ok_l1']),
123     Clause('lit_l2', ['light_l2', 'live_l2', 'ok_l2']),
124     Clause('live_l1', ['live_w0']),
125     Clause('live_w0', ['up_s2', 'live_w1']),
126     Clause('live_w0', ['down_s2', 'live_w2']),
127     Clause('live_w1', ['up_s3', 'live_w3']),
128     Clause('live_w2', ['down_s1', 'live_w3' ]),
129     Clause('live_l2', ['live_w4']),
130     Clause('live_w4', ['up_s3', 'live_w3' ]),
131     Clause('live_p_1', ['live_w3']),
132     Clause('live_w3', ['live_w5', 'ok_cb1']),
133     Askable('up_s1'),
134     Askable('down_s1'),
135     Askable('up_s2'),
136     Clause('light_l2'),
137     Clause('ok_l1'),
138     Clause('light_l2'),
139     Clause('ok_l1'),
140     Clause('ok_l2'),
141     Clause('ok_cb1'),
142     Clause('ok_cb2'),
143     Clause('live_outside'),
144     Clause('live_p_2', ['live_w6']),
145     Clause('live_w6', ['live_w5', 'ok_cb2']),
146     Clause('ok_l2'),
147     Clause('ok_cb1'),
148     Clause('ok_cb2'),
149     Clause('live_outside'),
150     Clause('live_p_2', ['live_w6']),
151     Clause('live_w6', ['live_w5', 'ok_cb2']),
152     Askable('down_s2'),
153     Askable('up_s3'),
154     Askable('down_s2')
155 ]
156
157 # 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

```

11 from logicProblem import yes
12
13 def fixed_point(kb):
14     """Returns the fixed point of knowledge base kb.
15     """
16     fp = ask_askables(kb)
17     added = True
18     while added:
19         added = False # added is true when an atom was added to fp this
20                        # iteration
21         for c in kb.clauses:
22             if c.head not in fp and all(b in fp for b in c.body):
23                 fp.add(c.head)
24                 added = True
25                 kb.display(2,c.head,"added to fp due to clause",c)
26     return fp
27
28 def ask_askables(kb):
29     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`:

```

30 from logicProblem import triv_KB
31 def test(kb=triv_KB, fixedpt = {'i_am','i_think'}):
32     fp = fixed_point(kb)
33     assert fp == fixedpt, f"kb gave result {fp}"
34     print("Passed unit test")
35 if __name__ == "__main__":
36     test()
37
38 from logicProblem import elect
39 # elect.max_display_level=3 # give detailed trace
40 # fixed_point(elect)

```

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

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

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

complexity as a function of n and b , the maximum number of atoms in the body of a clause?

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

```

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

```

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

```

29 from logicProblem import triv_KB
30 def test():
31     a1 = prove(triv_KB, ['i_am'])
32     assert a1, f"triv_KB proving i_am gave {a1}"
33     a2 = prove(triv_KB, ['i_smell'])
34     assert not a2, f"triv_KB proving i_smell gave {a2}"
35     print("Passed unit tests")
36 if __name__ == "__main__":
37     test()
38 # try
39 from logicProblem import elect

```



```

40 # elect.max_display_level=3 # give detailed trace
41 # prove(elect,['live_w6'])
42 # 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_____
11 from logicProblem import yes # for asking the user
12
13 def prove_atom(kb, atom, indent=""):
14     """returns a pair (atom,proofs) where proofs is the list of proofs
15     of the elements of a body of a clause used to prove atom.
16     """
17     kb.display(2,indent,'proving',atom)
18     if atom in kb.askables:
19         if yes(input("Is "+atom+" true? ")):
20             return (atom,"answered")
21         else:
22             return "fail"
23     else:
24         for cl in kb.clauses_for_atom(atom):
25             kb.display(2,indent,"trying",atom,'<-', ' & '.join(cl.body))
26             pr_body = prove_body(kb, cl.body, indent)
27             if pr_body != "fail":
28                 return (atom, pr_body)
29         return "fail"
30
31 def prove_body(kb, ans_body, indent=""):
32     """returns proof tree if kb |- ans_body or "fail" if there is no proof
33     ans_body is a list of atoms in a body to be proved
34     """
35     proofs = []
36     for atom in ans_body:

```

```

37     proof_at = prove_atom(kb, atom, indent+" ")
38     if proof_at == "fail":
39         return "fail" # fail if any proof fails
40     else:
41         proofs.append(proof_at)
42     return proofs

```

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

```

_____logicExplain.py — (continued)_____
44 from logicProblem import triv_KB
45 def test():
46     a1 = prove_atom(triv_KB, 'i_am')
47     assert a1, f"triv_KB proving i_am gave {a1}"
48     a2 = prove_atom(triv_KB, 'i_smell')
49     assert a2=="fail", "triv_KB proving i_smell gave {a2}"
50     print("Passed unit tests")
51 if __name__ == "__main__":
52     test()
53 # try
54 from logicProblem import elect, elect_bug
55 # elect.max_display_level=3 # give detailed trace
56 # prove_atom(elect, 'live_w6')
57 # 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)_____
59 helptext = """Commands are:
60 ask atom    ask is there is a proof for atom (atom should not be in quotes)
61 how        show the clause that was used to prove atom
62 how n      show the clause used to prove the nth element of the body
63 up         go back up proof tree to explore other parts of the proof tree
64 kb         print the knowledge base
65 quit       quit this interaction (and go back to Python)
66 help       print this text
67 """
68
69 def interact(kb):
70     going = True
71     ups = [] # stack for going up

```

```

72     proof="fail" # there is no proof to start
73     while going:
74         inp = input("logicExplain: ")
75         inps = inp.split(" ")
76         try:
77             command = inps[0]
78             if command == "quit":
79                 going = False
80             elif command == "ask":
81                 proof = prove_atom(kb, inps[1])
82                 if proof == "fail":
83                     print("fail")
84                 else:
85                     print("yes")
86             elif command == "how":
87                 if proof=="fail":
88                     print("there is no proof")
89                 elif len(inps)==1:
90                     print_rule(proof)
91                 else:
92                     try:
93                         ups.append(proof)
94                         proof = proof[1][int(inps[1])] #nth argument of rule
95                         print_rule(proof)
96                     except:
97                         print('In "how n", n must be a number between 0
98                             and',len(proof[1])-1,"inclusive.")
99             elif command == "up":
100                 if ups:
101                     proof = ups.pop()
102                 else:
103                     print("No rule to go up to.")
104                     print_rule(proof)
105             elif command == "kb":
106                 print(kb)
107             elif command == "help":
108                 print helptext
109             else:
110                 print("unknown command:", inp)
111                 print("use help for help")
112         except:
113             print("unknown command:", inp)
114             print("use help for help")
115
116 def print_rule(proof):
117     (head,body) = proof
118     if body == "answered":
119         print(head,"was answered yes")
120     elif body == []:
121         print(head,"is a fact")

```

```

121     else:
122         print(head,"<-")
123         for i,a in enumerate(body):
124             print(i,":",a[0])
125
126 # try
127 # interact(elect)
128 # Which clause is wrong in elect_bug? Try:
129 # interact(elect_bug)
130 # 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_____
11 from logicProblem import Clause, Askable, KB, yes
12
13 class Assumable(object):
14     """An askable atom"""
15
16     def __init__(self, atom):
17         """clause with atom head and lost of atoms body"""
18         self.atom = atom
19
20     def __str__(self):
21         """returns the string representation of a clause.
22         """
23         return "assumable " + self.atom + "."
24
25 class KBA(KB):
26     """A knowledge base that can include assumables"""
27     def __init__(self, statements):
28         self.assumables = [c.atom for c in statements if isinstance(c,
29                               Assumable)]
29         KB.__init__(self, statements)

```

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

```

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

```

```

48         for cl in self.clauses_for_atom(selected)
49         for ass in
            self.prove_all_ass(cl.body+ans_body[1:], assumed)
50         ] # union of answers for each clause with
            head=selected
51     else:
            # empty body
52     return [assumed] # one answer
53
54 def conflicts(self):
55     """returns a list of minimal conflicts"""
56     return minsets(self.prove_all_ass(['false']))

```

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

```

_____logicAssumables.py — (continued)_____
58 def minsets(ls):
59     """ls is a list of sets
60     returns a list of minimal sets in ls
61     """
62     ans = [] # elements known to be minimal
63     for c in ls:
64         if not any(c1<c for c1 in ls) and not any(c1 <= c for c1 in ans):
65             ans.append(c)
66     return ans
67
68 # minsets([{2, 3, 4}, {2, 3}, {6, 2, 3}, {2, 3}, {2, 4, 5}])

```

Warning: *minsets* works for a list of sets or for a set of (frozen) sets, but it does not work for a generator of sets (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)_____
69 def diagnoses(cons):
70     """cons is a list of (minimal) conflicts.
71     returns a list of diagnoses."""
72     if cons == []:
73         return [set()]
74     else:
75         return minsets([({e}|d) # | is set union
76                         for e in cons[0]
77                         for d in diagnoses(cons[1:])])

```

Test cases:

```

_____logicAssumables.py — (continued)_____
80 electa = KBA([

```

```

81     Clause('light_l1'),
82     Clause('light_l2'),
83     Assumable('ok_l1'),
84     Assumable('ok_l2'),
85     Assumable('ok_s1'),
86     Assumable('ok_s2'),
87     Assumable('ok_s3'),
88     Assumable('ok_cb1'),
89     Assumable('ok_cb2'),
90     Assumable('live_outside'),
91     Clause('live_l1', ['live_w0']),
92     Clause('live_w0', ['up_s2', 'ok_s2', 'live_w1']),
93     Clause('live_w0', ['down_s2', 'ok_s2', 'live_w2']),
94     Clause('live_w1', ['up_s1', 'ok_s1', 'live_w3']),
95     Clause('live_w2', ['down_s1', 'ok_s1', 'live_w3' ]),
96     Clause('live_l2', ['live_w4']),
97     Clause('live_w4', ['up_s3', 'ok_s3', 'live_w3' ]),
98     Clause('live_p_1', ['live_w3']),
99     Clause('live_w3', ['live_w5', 'ok_cb1']),
100    Clause('live_p_2', ['live_w6']),
101    Clause('live_w6', ['live_w5', 'ok_cb2']),
102    Clause('live_w5', ['live_outside']),
103    Clause('lit_l1', ['light_l1', 'live_l1', 'ok_l1']),
104    Clause('lit_l2', ['light_l2', 'live_l2', 'ok_l2']),
105    Askable('up_s1'),
106    Askable('down_s1'),
107    Askable('up_s2'),
108    Askable('down_s2'),
109    Askable('up_s3'),
110    Askable('down_s2'),
111    Askable('dark_l1'),
112    Askable('dark_l2'),
113    Clause('false', ['dark_l1', 'lit_l1']),
114    Clause('false', ['dark_l2', 'lit_l2'])
115    ])
116 # electa.prove_all_ass(['false'])
117 # cs=electa.conflicts()
118 # print(cs)
119 # diagnoses(cs)      # diagnoses from conflicts

```

Exercise 5.7 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 iter-

ative deepening on the number of assumptions, generating conflicts and explanations together, and pruning as early as possible.

5.6 Negation-as-failure

The negation of an atom a is written as $\text{Not}(a)$ in a body.

```

11 from logicProblem import KB, Clause, Askable, yes
12
13 class Not(object):
14     def __init__(self, atom):
15         self.theatom = atom
16
17     def atom(self):
18         return self.theatom
19
20     def __repr__(self):
21         return f"Not({self.theatom})"

```

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

```

23 def prove_naf(kb, ans_body, indent=""):
24     """ prove with negation-as-failure and askables
25     returns True if kb |- ans_body
26     ans_body is a list of atoms to be proved
27     """
28     kb.display(2,indent,'yes <-', ' & '.join(str(e) for e in ans_body))
29     if ans_body:
30         selected = ans_body[0] # select first atom from ans_body
31         if isinstance(selected, Not):
32             kb.display(2,indent,f"proving {selected.atom()}")
33             if prove_naf(kb, [selected.atom()], indent):
34                 kb.display(2,indent,f"{selected.atom()} succeeded so
35                     Not({selected.atom()}) fails")
36                 return False
37             else:
38                 kb.display(2,indent,f"{selected.atom()} fails so
39                     Not({selected.atom()}) succeeds")
40                 return prove_naf(kb, ans_body[1:],indent+" ")
41         if selected in kb.askables:
42             return (yes(input("Is "+selected+" true? "))
43                 and prove_naf(kb,ans_body[1:],indent+" "))
44         else:
45             return any(prove_naf(kb,cl.body+ans_body[1:],indent+" ")
46                 for cl in kb.clauses_for_atom(selected))
47     else:
48         return True # empty body is true

```


Test cases:

```

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

```

Default reasoning example:

```

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

```


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
11 class Strips(object):
12     def __init__(self, name, preconds, effects, cost=1):
13         """
14         defines the STRIPS representation for an action:
15         * name is the name of the action
16         * preconds, the preconditions, is feature:value dictionary that
           must hold
17         for the action to be carried out
18         * effects is a feature:value map that this action makes
19         true. The action changes the value of any feature specified
20         here, and leaves other features unchanged.
21         * cost is the cost of the action
22         """
```

```

23     self.name = name
24     self.preconds = preconds
25     self.effects = effects
26     self.cost = cost
27
28     def __repr__(self):
29         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)
31 class STRIPS_domain(object):
32     def __init__(self, feature_domain_dict, actions):
33         """Problem domain
34         feature_domain_dict is a feature:domain dictionary,
35         mapping each feature to its domain
36         actions
37         """
38         self.feature_domain_dict = feature_domain_dict
39         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)
41 class Planning_problem(object):
42     def __init__(self, prob_domain, initial_state, goal):
43         """
44         a planning problem consists of
45         * a planning domain
46         * the initial state
47         * a goal
48         """
49         self.prob_domain = prob_domain
50         self.initial_state = initial_state
51         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.

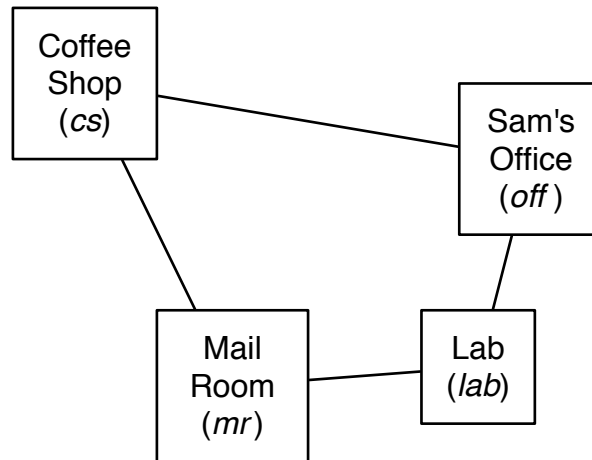
**Features to describe states***RLoc* – Rob's location*RHC* – Rob has coffee*SWC* – Sam wants coffee*MW* – Mail is waiting*RHM* – Rob has mail**Actions***mc* – move clockwise*mcc* – move counterclockwise*puc* – pickup coffee*dc* – deliver coffee*pum* – pickup mail*dm* – deliver mail

Figure 6.1: Robot Delivery Domain

stripsProblem.py — (continued)

```

53 boolean = {True, False}
54 delivery_domain = STRIPS_domain(
55     {'RLoc':{'cs', 'off', 'lab', 'mr'}, 'RHC':boolean, 'SWC':boolean,
56      'MW':boolean, 'RHM':boolean},      #feature:values dictionary
57     { Strips('mc_cs', {'RLoc':'cs'}, {'RLoc':'off'}),
58       Strips('mc_off', {'RLoc':'off'}, {'RLoc':'lab'}),
59       Strips('mc_lab', {'RLoc':'lab'}, {'RLoc':'mr'}),
60       Strips('mc_mr', {'RLoc':'mr'}, {'RLoc':'cs'}),
61       Strips('mcc_cs', {'RLoc':'cs'}, {'RLoc':'mr'}),
62       Strips('mcc_off', {'RLoc':'off'}, {'RLoc':'cs'}),
63       Strips('mcc_lab', {'RLoc':'lab'}, {'RLoc':'off'}),
64       Strips('mcc_mr', {'RLoc':'mr'}, {'RLoc':'lab'}),
65       Strips('puc', {'RLoc':'cs', 'RHC':False}, {'RHC':True}),
66       Strips('dc', {'RLoc':'off', 'RHC':True}, {'RHC':False, 'SWC':False}),
67       Strips('pum', {'RLoc':'mr', 'MW':True}, {'RHM':True, 'MW':False}),
68       Strips('dm', {'RLoc':'off', 'RHM':True}, {'RHM':False})
69     } )

```

stripsProblem.py — (continued)

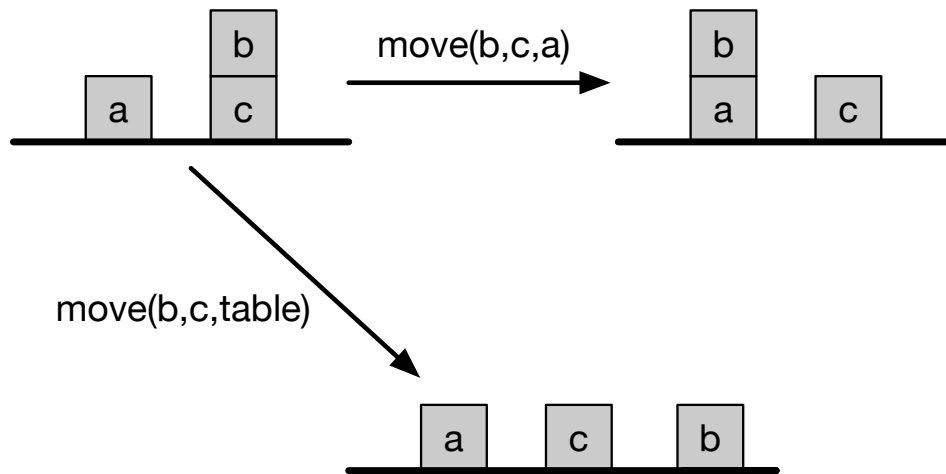


Figure 6.2: Blocks world with two actions

```

71 | problem0 = Planning_problem(delivery_domain,
72 |                             {'RLoc': 'lab', 'MW': True, 'SWC': True, 'RHC': False,
73 |                             'RHM': False},
74 |                             {'RLoc': 'off'})
75 | problem1 = Planning_problem(delivery_domain,
76 |                             {'RLoc': 'lab', 'MW': True, 'SWC': True, 'RHC': False,
77 |                             'RHM': False},
78 |                             {'SWC': False})
79 | problem2 = Planning_problem(delivery_domain,
80 |                             {'RLoc': 'lab', 'MW': True, 'SWC': True, 'RHC': False,
81 |                             'RHM': False},
82 |                             {'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)
84  """ blocks world
85  def move(x,y,z):
86      """string for the 'move' action"""
87      return 'move_'+x+'_from_'+y+'_to_'+z
88  def on(x):
89      """string for the 'on' feature"""
90      return x+'_is_on'
91  def clear(x):
92      """string for the 'clear' feature"""
93      return 'clear_'+x
94  def create_blocks_world(blocks = {'a','b','c','d'}):
95      blocks_and_table = blocks | {'table'}
96      stmap = {Strips(move(x,y,z),{on(x):y, clear(x):True, clear(z):True},
97                      {on(x):z, clear(y):True, clear(z):False})
98              for x in blocks
99              for y in blocks_and_table
100             for z in blocks
101             if x!=y and y!=z and z!=x}
102      stmap.update({Strips(move(x,y,'table'), {on(x):y, clear(x):True},
103                  {on(x):'table', clear(y):True})
104                  for x in blocks
105                  for y in blocks
106                  if x!=y})
107      feature_domain_dict = {on(x):blocks_and_table-{x} for x in blocks}
108      feature_domain_dict.update({clear(x):boolean for x in blocks_and_table})
109      return STRIPS_domain(feature_domain_dict, stmap)

```

The problem *blocks1* 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)
111 blocks1dom = create_blocks_world({'a','b','c'})
112 blocks1 = Planning_problem(blocks1dom,
113     {on('a'):'table', clear('a'):True,
114     on('b'):'c', clear('b'):True,
115     on('c'):'table', clear('c'):False}, # initial state
116     {on('a'):'b', on('c'):'a'}) #goal

```

The problem *blocks2* is one to invert a tower of size 4.

```

stripsProblem.py — (continued)
118 blocks2dom = create_blocks_world({'a','b','c','d'})

```

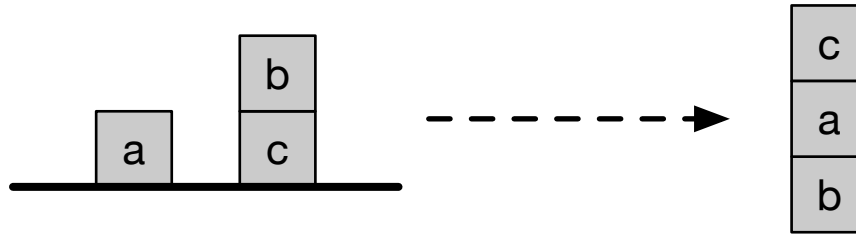


Figure 6.3: Blocks problem blocks1

```

119 tower4 = {clear('a'):True, on('a'):'b',
120           clear('b'):False, on('b'):'c',
121           clear('c'):False, on('c'):'d',
122           clear('d'):False, on('d'):'table'}
123 blocks2 = Planning_problem(blocks2dom,
124                             tower4, # initial state
125                             {on('d'):'c', on('c'):'b', on('b'):'a'}) #goal

```

The problem *blocks3* is to move the bottom block to the top of a tower of size 4.

```

stripsProblem.py — (continued)
127 blocks3 = Planning_problem(blocks2dom,
128                             tower4, # initial state
129                             {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
11 from searchProblem import Arc, Search_problem
12 from stripsProblem import Strips, STRIPS_domain
13
14 class State(object):
15     def __init__(self, assignment):
16         self.assignment = assignment
17         self.hash_value = None
18     def __hash__(self):
19         if self.hash_value is None:
20             self.hash_value = hash(frozenset(self.assignment.items()))
21         return self.hash_value
22     def __eq__(self, st):
23         return self.assignment == st.assignment
24     def __str__(self):
25         return str(self.assignment)

```

In order to define a search problem (page 37), we need to define the goal condition, the start nodes, the neighbours, and (optionally) a heuristic function. Here *zero* is the default heuristic function.

```

stripsForwardPlanner.py — (continued)
27 def zero(*args,**nargs):
28     """always returns 0"""
29     return 0
30
31 class Forward_STRIPS(Search_problem):
32     """A search problem from a planning problem where:
33     * a node is a state object.
34     * the dynamics are specified by the STRIPS representation of actions
35     """
36     def __init__(self, planning_problem, heur=zero):
37         """creates a forward search space from a planning problem.
38         heur(state,goal) is a heuristic function,
39         an underestimate of the cost from state to goal, where
40         both state and goals are feature:value dictionaries.
41         """
42         self.prob_domain = planning_problem.prob_domain
43         self.initial_state = State(planning_problem.initial_state)
44         self.goal = planning_problem.goal
45         self.heur = heur
46
47     def is_goal(self, state):
48         """is True if node is a goal.
49
50         Every goal feature has the same value in the state and the goal."""
51         return all(state.assignment[prop]==self.goal[prop]

```

```

52         for prop in self.goal)
53
54     def start_node(self):
55         """returns start node"""
56         return self.initial_state
57
58     def neighbors(self, state):
59         """returns neighbors of state in this problem"""
60         return [ Arc(state, self.effect(act, state.assignment), act.cost,
61                     act)
62                 for act in self.prob_domain.actions
63                 if self.possible(act, state.assignment)]
64
65     def possible(self, act, state_asst):
66         """True if act is possible in state.
67         act is possible if all of its preconditions have the same value in
68         the state"""
69         return all(state_asst[pre] == act.preconds[pre]
70                   for pre in act.preconds)
71
72     def effect(self, act, state_asst):
73         """returns the state that is the effect of doing act given
74         state_asst
75         Python 3.9: return state_asst | act.effects"""
76         new_state_asst = state_asst.copy()
77         new_state_asst.update(act.effects)
78         return State(new_state_asst)
79
80     def heuristic(self, state):
81         """in the forward planner a node is a state.
82         the heuristic is an (under)estimate of the cost
83         of going from the state to the top-level goal.
84         """
85         return self.heur(state.assignment, self.goal)

```

Here are some test cases to try.

```

stripsForwardPlanner.py — (continued)
84 from searchBranchAndBound import DF_branch_and_bound
85 from searchMPP import SearcherMPP
86 from stripsProblem import problem0, problem1, problem2, blocks1, blocks2,
87     blocks3
88
89 # SearcherMPP(Forward_STRIPS(problem1)).search() #A* with MPP
90 # DF_branch_and_bound(Forward_STRIPS(problem1),10).search() #B&B
91 # To find more than one plan:
92 # s1 = SearcherMPP(Forward_STRIPS(problem1)) #A*
93 # 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
11 def dist(loc1, loc2):
12     """returns the distance from location loc1 to loc2
13     """
14     if loc1==loc2:
15         return 0
16     if {loc1,loc2} in [{'cs','lab'},{'mr','off'}]:
17         return 2
18     else:
19         return 1

```

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

```

stripsHeuristic.py — (continued)
21 def h1(state,goal):
22     """ the distance to the goal location, if there is one"""
23     if 'RLoc' in goal:
24         return dist(state['RLoc'], goal['RLoc'])
25     else:
26         return 0
27
28 def h2(state,goal):
29     """ the distance to the coffee shop plus getting coffee and delivering
30     it
31     if the robot needs to get coffee
32     """
33     if ('SWC' in goal and goal['SWC']==False
34         and state['SWC']==True
35         and state['RHC']==False):
36         return dist(state['RLoc'],'cs')+3
37     else:
38         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)
39 def maxh(*heuristics):
40     """Returns a new heuristic function that is the maximum of the
        functions in heuristics.
41     heuristics is the list of arguments which must be heuristic functions.
42     """
43     # return lambda state,goal: max(h(state,goal) for h in heuristics)
44     def newh(state,goal):
45         return max(h(state,goal) for h in heuristics)
46     return newh

```

The following runs the example with and without the heuristic.

```

stripsHeuristic.py — (continued)
48 ##### Forward Planner #####
49 from searchMPP import SearcherMPP
50 from stripsForwardPlanner import Forward_STRIPS
51 from stripsProblem import problem0, problem1, problem2, blocks1, blocks2,
    blocks3
52
53 def test_forward_heuristic(thisproblem=problem1):
54     print("\n***** FORWARD NO HEURISTIC")
55     print(SearcherMPP(Forward_STRIPS(thisproblem)).search())
56
57     print("\n***** FORWARD WITH HEURISTIC h1")
58     print(SearcherMPP(Forward_STRIPS(thisproblem,h1)).search())
59
60     print("\n***** FORWARD WITH HEURISTIC h2")
61     print(SearcherMPP(Forward_STRIPS(thisproblem,h2)).search())
62
63     print("\n***** FORWARD WITH HEURISTICS h1 and h2")
64     print(SearcherMPP(Forward_STRIPS(thisproblem,maxh(h1,h2))).search())
65
66 if __name__ == "__main__":
67     test_forward_heuristic()

```

Exercise 6.4 For more than one start-state/goal combination, test the forward planner with a heuristic function of just h_1 , with just h_2 and with both. Explain why each one prunes or doesn't prune the search space.

Exercise 6.5 Create a better heuristic than $\max h(h_1, h_2)$. Try it for a number of different problems. In particular, try and include the following costs:

- i) h_3 is like h_2 but also takes into account the case when R_{loc} is in goal.
- ii) h_4 uses the distance to the mail room plus getting mail and delivering it if the robot needs to get need to deliver mail.
- iii) h_5 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
11 from searchProblem import Arc, Search_problem
12
13 class Subgoal(object):
14     def __init__(self, assignment):
15         self.assignment = assignment
16         self.hash_value = None
17     def __hash__(self):
18         if self.hash_value is None:
19             self.hash_value = hash(frozenset(self.assignment.items()))
20         return self.hash_value
21     def __eq__(self, st):
22         return self.assignment == st.assignment
23     def __str__(self):
24         return str(self.assignment)

```

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

```

stripsRegressionPlanner.py — (continued)
26 from stripsForwardPlanner import zero
27
28 class Regression_STRIPS(Search_problem):
29     """A search problem where:
30     * a node is a goal to be achieved, represented by a set of propositions.
31     * the dynamics are specified by the STRIPS representation of actions
32     """
33
34     def __init__(self, planning_problem, heur=zero):
35         """creates a regression search space from a planning problem.
36         heur(state,goal) is a heuristic function;
37         an underestimate of the cost from state to goal, where
38         both state and goals are feature:value dictionaries
39         """
40         self.prob_domain = planning_problem.prob_domain
41         self.top_goal = Subgoal(planning_problem.goal)
42         self.initial_state = planning_problem.initial_state
43         self.heur = heur

```

```

44
45 def is_goal(self, subgoal):
46     """if subgoal is true in the initial state, a path has been found"""
47     goal_asst = subgoal.assignment
48     return all(self.initial_state[g]==goal_asst[g]
49               for g in goal_asst)
50
51 def start_node(self):
52     """the start node is the top-level goal"""
53     return self.top_goal
54
55 def neighbors(self, subgoal):
56     """returns a list of the arcs for the neighbors of subgoal in this
57     problem"""
58     goal_asst = subgoal.assignment
59     return [ Arc(subgoal, self.weakest_precond(act, goal_asst),
60               act.cost, act)
61             for act in self.prob_domain.actions
62             if self.possible(act, goal_asst)]
63
64 def possible(self, act, goal_asst):
65     """True if act is possible to achieve goal_asst.
66
67     the action achieves an element of the effects and
68     the action doesn't delete something that needs to be achieved and
69     the preconditions are consistent with other subgoals that need to
70     be achieved
71     """
72     return ( any(goal_asst[prop] == act.effects[prop]
73               for prop in act.effects if prop in goal_asst)
74             and all(goal_asst[prop] == act.effects[prop]
75               for prop in act.effects if prop in goal_asst)
76             and all(goal_asst[prop] == act.preconds[prop]
77               for prop in act.preconds if prop not in act.effects
78               and prop in goal_asst)
79             )
80
81 def weakest_precond(self, act, goal_asst):
82     """returns the subgoal that must be true so goal_asst holds after
83     act
84     should be: act.preconds | (goal_asst - act.effects)
85     """
86     new_asst = act.preconds.copy()
87     for g in goal_asst:
88         if g not in act.effects:
89             new_asst[g] = goal_asst[g]
90     return Subgoal(new_asst)
91
92 def heuristic(self, subgoal):
93     """in the regression planner a node is a subgoal.

```

```

89         the heuristic is an (under)estimate of the cost of going from the
          initial state to subgoal.
90         """
91         return self.heur(self.initial_state, subgoal.assignment)

```

```

_____stripsRegressionPlanner.py — (continued) _____
93 from searchBranchAndBound import DF_branch_and_bound
94 from searchMPP import SearcherMPP
95 from stripsProblem import problem0, problem1, problem2, blocks1, blocks2,
   blocks3
96
97 # SearcherMPP(Regression_STRIPS(problem1)).search() #A* with MPP
98 # 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 $\{a : \text{True}, b : \text{False}\}$ has been visited, then any node that is a superset, e.g., $\{a : \text{True}, b : \text{False}, d : \text{True}\}$, need not be expanded. If the simpler subgoal does not lead to a solution, the more complicated one won't either. Implement this more severe pruning. (Hint: This may require modifications to the searcher.)

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

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

6.3.1 Defining Heuristics for a Regression Planner

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

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

```

_____stripsHeuristic.py — (continued) _____
69 ##### Regression Planner
70 from stripsRegressionPlanner import Regression_STRIPS

```

```

71
72 def test_regression_heuristic(thisproblem=problem1):
73     print("\n***** REGRESSION NO HEURISTIC")
74     print(SearcherMPP(Regression_STRIPS(thisproblem)).search())
75
76     print("\n***** REGRESSION WITH HEURISTICS h1 and h2")
77     print(SearcherMPP(Regression_STRIPS(thisproblem,maxh(h1,h2))).search())
78
79 if __name__ == "__main__":
80     test_regression_heuristic()

```

Exercise 6.10 Try the regression planner with a heuristic function of just 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_____
11 from cspProblem import Variable, CSP, Constraint
12
13 class CSP_from_STRIPS(CSP):
14     """A CSP where:
15     * CSP variables are constructed for each feature and time, and each
      action and time
16     * the dynamics are specified by the STRIPS representation of actions
17     """
18
19     def __init__(self, planning_problem, number_stages=2):
20         prob_domain = planning_problem.prob_domain
21         initial_state = planning_problem.initial_state
22         goal = planning_problem.goal
23         # self.action_vars[t] is the action variable for time t
24         self.action_vars = [Variable(f"Action{t}", prob_domain.actions)
25                             for t in range(number_stages)]
26         # feat_time_var[f][t] is the variable for feature f at time t
27         feat_time_var = {feat: [Variable(f"{feat}_{t}", dom)
28                                   for t in range(number_stages+1)]

```



```

29         for (feat,dom) in
30             prob_domain.feature_domain_dict.items()}
31
32     # initial state constraints:
33     constraints = [Constraint((feat_time_var[feat][0],), is_(val))
34                       for (feat,val) in initial_state.items())
35
36     # goal constraints on the final state:
37     constraints += [Constraint((feat_time_var[feat][number_stages],),
38                               is_(val))
39                     for (feat,val) in goal.items())
40
41     # precondition constraints:
42     constraints += [Constraint((feat_time_var[feat][t],
43                               self.action_vars[t]),
44                               if_(val,act)) # feat@t==val if action@t==act
45                               for act in prob_domain.actions
46                               for (feat,val) in act.preconds.items()
47                               for t in range(number_stages)]
48
49     # effect constraints:
50     constraints += [Constraint((feat_time_var[feat][t+1],
51                               self.action_vars[t]),
52                               if_(val,act)) # feat@t+1==val if
53                               action@t==act
54                               for act in prob_domain.actions
55                               for feat,val in act.effects.items()
56                               for t in range(number_stages)]
57
58     # frame constraints:
59     constraints += [Constraint((feat_time_var[feat][t],
60                               self.action_vars[t], feat_time_var[feat][t+1]),
61                               eq_if_not_in_({act for act in
62                                               prob_domain.actions
63                                               if feat in act.effects}))
64                               for feat in prob_domain.feature_domain_dict
65                               for t in range(number_stages) ]
66
67     variables = set(self.action_vars) | {feat_time_var[feat][t]
68                                         for feat in
69                                             prob_domain.feature_domain_dict
70                                             for t in range(number_stages+1)}
71
72     CSP.__init__(self, variables, constraints)
73
74     def extract_plan(self,soln):
75         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)
68 def is_(val):
69     """returns a function that is true when it is applied to val.
70     """
71     #return lambda x: x == val
72     def is_fun(x):
73         return x == val
74     is_fun.__name__ = f"value_is_{val}"
75     return is_fun
76
77 def if_(v1,v2):
78     """if the second argument is v2, the first argument must be v1"""
79     #return lambda x1,x2: x1==v1 if x2==v2 else True
80     def if_fun(x1,x2):
81         return x1==v1 if x2==v2 else True
82     if_fun.__name__ = f"if x2 is {v2} then x1 is {v1}"
83     return if_fun
84
85 def eq_if_not_in_(actset):
86     """first and third arguments are equal if action is not in actset"""
87     # return lambda x1, a, x2: x1==x2 if a not in actset else True
88     def eq_if_not_fun(x1, a, x2):
89         return x1==x2 if a not in actset else True
90     eq_if_not_fun.__name__ = f"first and third arguments are equal if
        action is not in {actset}"
91     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)
93 def con_plan(prob,horizon):
94     """finds a plan for problem prob given horizon.
95     """
96     csp = CSP_from_STRIPS(prob, horizon)
97     sol = Con_solver(csp).solve_one()
98     return csp.extract_plan(sol) if sol else sol

```

The following are some example queries.

```

stripsCSPPlanner.py — (continued)
100 from searchGeneric import Searcher

```

```

101 from stripsProblem import delivery_domain
102 from cspConsistency import Search_with_AC_from_CSP, Con_solver
103 from stripsProblem import Planning_problem, problem0, problem1, problem2,
    blocks1, blocks2, blocks3
104
105 # Problem 0
106 # con_plan(problem0,1) # should it succeed?
107 # con_plan(problem0,2) # should it succeed?
108 # con_plan(problem0,3) # should it succeed?
109 # To use search to enumerate solutions
110 #searcher0a = Searcher(Search_with_AC_from_CSP(CSP_from_STRIPS(problem0,
    1)))
111 #print(searcher0a.search()) # returns path to solution
112
113 ## Problem 1
114 # con_plan(problem1,5) # should it succeed?
115 # con_plan(problem1,4) # should it succeed?
116 ## To use search to enumerate solutions:
117 #searcher15a = Searcher(Search_with_AC_from_CSP(CSP_from_STRIPS(problem1,
    5)))
118 #print(searcher15a.search()) # returns path to solution
119
120 ## Problem 2
121 #con_plan(problem2, 6) # should fail??
122 #con_plan(problem2, 7) # should succeed???
123
124 ## Example 6.13
125 problem3 = Planning_problem(delivery_domain,
126                             {'SWC':True, 'RHC':False}, {'SWC':False})
127 #con_plan(problem3,2) # Horizon of 2
128 #con_plan(problem3,3) # Horizon of 3
129
130 problem4 = Planning_problem(delivery_domain,{'SWC':True},
131                             {'SWC':False, 'MW':False, 'RHM':False})
132
133 # For the stochastic local search:
134 #from cspSLS import SLSearcher, Runtime_distribution
135 # cspplanning15 = CSP_from_STRIPS(problem1, 5) # should succeed
136 #se0 = SLSearcher(cspplanning15); print(se0.search(100000,0.5))
137 #p = Runtime_distribution(cspplanning15)
138 #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
11 from searchProblem import Arc, Search_problem
12 import random
13
14 class Action_instance(object):
15     next_index = 0
16     def __init__(self, action, index=None):
17         if index is None:
18             index = Action_instance.next_index
19             Action_instance.next_index += 1
20         self.action = action
21         self.index = index
22
23     def __str__(self):
24         return f"{self.action}#{self.index}"
25
26     __repr__ = __str__ # __repr__ function is the same as the __str__
                        function

```

A node (as in the abstraction of search space) in a partial-order planner consists of:

- *actions*: a set of action instances.
- *constraints*: a set of (a_1, a_2) pairs, where a_1 and a_2 are action instances, which represents that a_1 must come before a_2 in the partial order. There are a number of ways that this could be represented. Here we represent the set of pairs that are in transitive closure of the *before* relation. This lets us quickly determine whether some before relation is consistent with the current constraints.
- *agenda*: a list of (s, a) pairs, where s is a (var, val) pair and a is an action instance. This means that variable var must have value val before a can occur.
- *causal_links*: a set of (a_0, g, a_1) triples, where a_1 and a_2 are action instances and g is a (var, val) pair. This holds when action a_0 makes g true for action a_1 .

```

stripsPOP.py — (continued)
28 class POP_node(object):
29     """a (partial) partial-order plan. This is a node in the search
        space."""
30     def __init__(self, actions, constraints, agenda, causal_links):
31         """

```

```

32     * actions is a set of action instances
33     * constraints a set of (a0,a1) pairs, representing a0<a1,
34       closed under transitivity
35     * agenda list of (subgoal,action) pairs to be achieved, where
36       subgoal is a (variable,value) pair
37     * causal_links is a set of (a0,g,a1) triples,
38       where ai are action instances, and g is a (variable,value) pair
39     """
40     self.actions = actions # a set of action instances
41     self.constraints = constraints # a set of (a0,a1) pairs
42     self.agenda = agenda # list of (subgoal,action) pairs to be
        achieved
43     self.causal_links = causal_links # set of (a0,g,a1) triples
44
45     def __str__(self):
46         return ("actions: "+str({str(a) for a in self.actions})+
47             "\nconstraints: "+
48             str({(str(a1),str(a2)) for (a1,a2) in self.constraints})+
49             "\nagenda: "+
50             str([(str(s),str(a)) for (s,a) in self.agenda])+
51             "\ncausal_links:"+
52             str({(str(a0),str(g),str(a2)) for (a0,g,a2) in
                self.causal_links}) )

```

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

```

stripsPOP.py — (continued)
54     def extract_plan(self):
55         """returns a total ordering of the action instances consistent
56         with the constraints.
57         raises IndexError if there is no choice.
58         """
59         sorted_acts = []
60         other_acts = set(self.actions)
61         while other_acts:
62             a = random.choice([a for a in other_acts if
63                 all(((a1,a) not in self.constraints) for a1 in
                    other_acts)])
64             sorted_acts.append(a)
65             other_acts.remove(a)
66         return sorted_acts

```

POP_search_from_STRIPS is an instance of a search problem. As such, we need to define the start nodes, the goal, and the neighbors of a node.

```

stripsPOP.py — (continued)
68 from display import Displayable
69
70 class POP_search_from_STRIPS(Search_problem, Displayable):
71     def __init__(self,planning_problem):

```

```

72     Search_problem.__init__(self)
73     self.planning_problem = planning_problem
74     self.start = Action_instance("start")
75     self.finish = Action_instance("finish")
76
77     def is_goal(self, node):
78         return node.agenda == []
79
80     def start_node(self):
81         constraints = {(self.start, self.finish)}
82         agenda = [(g, self.finish) for g in
83                 self.planning_problem.goal.items()]
83         return POP_node([self.start,self.finish], constraints, agenda, [] )

```

The *neighbors* method is a coroutine that enumerates the neighbors of a given node.

stripsPOP.py — (continued)

```

85     def neighbors(self, node):
86         """enumerates the neighbors of node"""
87         self.display(3,"finding neighbors of\n",node)
88         if node.agenda:
89             subgoal,act1 = node.agenda[0]
90             self.display(2,"selecting",subgoal,"for",act1)
91             new_agenda = node.agenda[1:]
92             for act0 in node.actions:
93                 if (self.achieves(act0, subgoal) and
94                     self.possible((act0,act1),node.constraints)):
95                     self.display(2," reusing",act0)
96                     consts1 =
97                         self.add_constraint((act0,act1),node.constraints)
98                     new_clink = (act0,subgoal,act1)
99                     new_cls = node.causal_links + [new_clink]
100                     for consts2 in
101                         self.protect_cl_for_actions(node.actions,consts1,new_clink):
102                         yield Arc(node,
103                             POP_node(node.actions,consts2,new_agenda,new_cls),
104                             cost=0)
105             for a0 in self.planning_problem.prob_domain.actions: #a0 is an
106                 action
107                 if self.achieves(a0, subgoal):
108                     #a0 acheieves subgoal
109                     new_a = Action_instance(a0)
110                     self.display(2," using new action",new_a)
111                     new_actions = node.actions + [new_a]
112                     consts1 =
113                         self.add_constraint((self.start,new_a),node.constraints)
114                     consts2 = self.add_constraint((new_a,act1),consts1)
115                     new_agenda1 = new_agenda + [(pre,new_a) for pre in
116                         a0.preconds.items()]
117                     new_clink = (new_a,subgoal,act1)

```

```

113         new_cls = node.causal_links + [new_clink]
114         for consts3 in
            self.protect_all_cls(node.causal_links,new_a,consts2):
115             for consts4 in
                self.protect_cl_for_actions(node.actions,consts3,new_clink):
116                 yield Arc(node,
117                             POP_node(new_actions,consts4,new_agenda1,new_cls),
118                             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)
120 def protect_cl_for_actions(self, actions, constrs, clink):
121     """yields constraints that extend constrs and
122     protect causal link (a0, subgoal, a1)
123     for each action in actions
124     """
125     if actions:
126         a = actions[0]
127         rem_actions = actions[1:]
128         a0, subgoal, a1 = clink
129         if a != a0 and a != a1 and self.deletes(a,subgoal):
130             if self.possible((a,a0),constrs):
131                 new_const = self.add_constraint((a,a0),constrs)
132                 for e in
                    self.protect_cl_for_actions(rem_actions,new_const,clink):
                        yield e # could be "yield from"
133             if self.possible((a1,a),constrs):
134                 new_const = self.add_constraint((a1,a),constrs)
135                 for e in
                    self.protect_cl_for_actions(rem_actions,new_const,clink):
                        yield e
136         else:
137             for e in
                self.protect_cl_for_actions(rem_actions,constrs,clink):
                    yield e
138     else:
139         yield constrs

```

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

```

stripsPOP.py — (continued)
141 def protect_all_cls(self, clinks, act, constrs):
142     """yields constraints that protect all causal links from act"""
143     if clinks:

```

```

144     (a0,cond,a1) = clinks[0] # select a causal link
145     rem_clinks = clinks[1:] # remaining causal links
146     if act != a0 and act != a1 and self.deletes(act,cond):
147         if self.possible((act,a0),constrs):
148             new_const = self.add_constraint((act,a0),constrs)
149             for e in self.protect_all_cls(rem_clinks,act,new_const):
150                 yield e
151         if self.possible((a1,act),constrs):
152             new_const = self.add_constraint((a1,act),constrs)
153             for e in self.protect_all_cls(rem_clinks,act,new_const):
154                 yield e
155     else:
156         for e in self.protect_all_cls(rem_clinks,act,constrs): yield
            e
157 else:
158     yield constrs

```

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

```

stripsPOP.py — (continued)
158 def achieves(self,action,subgoal):
159     var,val = subgoal
160     return var in self.effects(action) and self.effects(action)[var] ==
        val
161
162 def deletes(self,action,subgoal):
163     var,val = subgoal
164     return var in self.effects(action) and self.effects(action)[var] !=
        val
165
166 def effects(self,action):
167     """returns the variable:value dictionary of the effects of action.
168     works for both actions and action instances"""
169     if isinstance(action, Action_instance):
170         action = action.action
171     if action == "start":
172         return self.planning_problem.initial_state
173     elif action == "finish":
174         return {}
175     else:
176         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)
178 def add_constraint(self, pair, const):
179     if pair in const:

```



```

180         return const
181     todo = [pair]
182     newconst = const.copy()
183     while todo:
184         x0,x1 = todo.pop()
185         newconst.add((x0,x1))
186         for x,y in newconst:
187             if x==x1 and (x0,y) not in newconst:
188                 todo.append((x0,y))
189             if y==x0 and (x,x1) not in newconst:
190                 todo.append((x,x1))
191     return newconst
192
193 def possible(self,pair,constraint):
194     (x,y) = pair
195     return (y,x) not in constraint

```

Some code for testing:

```

stripsPOP.py — (continued)
197 from searchBranchAndBound import DF_branch_and_bound
198 from searchMPP import SearcherMPP
199 from stripsProblem import problem0, problem1, problem2, blocks1, blocks2,
    blocks3
200
201 rplanning0 = POP_search_from_STRIPS(problem0)
202 rplanning1 = POP_search_from_STRIPS(problem1)
203 rplanning2 = POP_search_from_STRIPS(problem2)
204 searcher0 = DF_branch_and_bound(rplanning0,5)
205 searcher0a = SearcherMPP(rplanning0)
206 searcher1 = DF_branch_and_bound(rplanning1,10)
207 searcher1a = SearcherMPP(rplanning1)
208 searcher2 = DF_branch_and_bound(rplanning2,10)
209 searcher2a = SearcherMPP(rplanning2)
210 # Try one of the following searchers
211 # a = searcher0.search()
212 # a = searcher0a.search()
213 # a.end().extract_plan() # print a plan found
214 # a.end().constraints # print the constraints
215 # SearcherMPP.max_display_level = 0 # less detailed display
216 # DF_branch_and_bound.max_display_level = 0 # less detailed display
217 # a = searcher1.search()
218 # a = searcher1a.search()
219 # a = searcher2.search()
220 # a = searcher2a.search()

```


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 networks 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 (car-book 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
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 $f.type$ is inferred from the $f.frange$ 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
11 import math, random, statistics
12 import csv
13 from display import Displayable
14 from utilities import argmax
15
16 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)
18 class Data_set(Displayable):
19     """ A dataset consists of a list of training data and a list of test
20         data.
21         """
22     def __init__(self, train, test=None, prob_test=0.20, target_index=0,
23                 header=None, target_type= None, seed=None): #12345):
24         """A dataset for learning.
25         train is a list of tuples representing the training examples
26         test is the list of tuples representing the test examples
27         if test is None, a test set is created by selecting each
28             example with probability prob_test
29         target_index is the index of the target.
30             If negative, it counts from right.
31             If target_index is larger than the number of properties,
32             there is no target (for unsupervised learning)
33         header is a list of names for the features
34         target_type is either None for automatic detection of target type
35             or one of "numeric", "boolean", "cartegorical"
36         seed is for random number; None gives a different test set each time
37         """
38         if seed: # given seed makes partition consistent from run-to-run
39             random.seed(seed)
40         if test is None:
41             train,test = partition_data(train, prob_test)
42         self.train = train
43         self.test = test
44
45         self.display(1,"Training set has",len(train),"examples. Number of
46             columns: ",{len(e) for e in train})
47         self.display(1,"Test set has",len(test),"examples. Number of
48             columns: ",{len(e) for e in test})
49         self.prob_test = prob_test
50         self.num_properties = len(self.train[0])
51         if target_index < 0: #allows for -1, -2, etc.
52             self.target_index = self.num_properties + target_index
53         else:
54             self.target_index = target_index
55         self.header = header
56         self.domains = [set() for i in range(self.num_properties)]
57         for example in self.train:
58             for ind,val in enumerate(example):
59                 self.domains[ind].add(val)
60         self.conditions_cache = {} # cache for computed conditions

```

```

59     self.create_features()
60     if target_type:
61         self.target.ftype = target_type
62     self.display(1, "There are", len(self.input_features), "input
        features")
63
64     def __str__(self):
65         if self.train and len(self.train)>0:
66             return ("Data: "+str(len(self.train))+ " training examples, "
67                     +str(len(self.test))+ " test examples, "
68                     +str(len(self.train[0]))+" features.")
69         else:
70             return ("Data: "+str(len(self.train))+ " training examples, "
71                     +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)
73     def create_features(self):
74         """create the set of features
75         """
76         self.target = None
77         self.input_features = []
78         for i in range(self.num_properties):
79             def feat(e, index=i):
80                 return e[index]
81             if self.header:
82                 feat.__doc__ = self.header[i]
83             else:
84                 feat.__doc__ = "e["+str(i)+"]"
85             feat.frange = list(self.domains[i])
86             feat.ftype = self.infer_type(feat.frange)
87             if i == self.target_index:
88                 self.target = feat
89             else:
90                 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)
92     def infer_type(self, domain):
93         """Infers the type of a feature with domain
94         """
95         if all(v in {True, False} for v in domain):
96             return "boolean"
97         if all(isinstance(v, (float, int)) for v in domain):
98             return "numeric"

```

```

99         else:
100             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)
102     def conditions(self, max_num_cuts=8, categorical_only = False):
103         """returns a set of boolean conditions from the input features
104         max_num_cuts is the maximum number of cuts for numeric features
105         categorical_only is true if only categorical features are made
106         binary
107         """
108         if (max_num_cuts, categorical_only) in self.conditions_cache:
109             return self.conditions_cache[(max_num_cuts, categorical_only)]
110         conds = []
111         for ind, frange in enumerate(self.domains):
112             if ind != self.target_index and len(frange)>1:
113                 if len(frange) == 2:
114                     # two values, the feature is equality to one of them.
115                     true_val = list(frange)[1] # choose one as true
116                     def feat(e, i=ind, tv=true_val):
117                         return e[i]==tv

```

```

117         if self.header:
118             feat.__doc__ = f"{self.header[ind]}=={true_val}"
119         else:
120             feat.__doc__ = f"e[{ind}]=={true_val}"
121         feat.frange = boolean
122         feat.ftype = "boolean"
123         conds.append(feat)
124     elif all(isinstance(val,(int,float)) for val in frange):
125         if categorical_only: # numeric, don't make cuts
126             def feat(e, i=ind):
127                 return e[i]
128             feat.__doc__ = f"e[{ind}]"
129             conds.append(feat)
130         else:
131             # all numeric, create cuts of the data
132             sorted_frange = sorted(frange)
133             num_cuts = min(max_num_cuts, len(frange))
134             cut_positions = [len(frange)*i//num_cuts for i in
135                             range(1,num_cuts)]
136             for cut in cut_positions:
137                 cutat = sorted_frange[cut]
138                 def feat(e, ind_=ind, cutat=cutat):
139                     return e[ind_] < cutat
140
141                 if self.header:
142                     feat.__doc__ = self.header[ind]+"<" +str(cutat)
143                 else:
144                     feat.__doc__ = "e["+str(ind)+"("<" +str(cutat)
145                 feat.frange = boolean
146                 feat.ftype = "boolean"
147                 conds.append(feat)
148     else:
149         # create an indicator function for every value
150         for val in frange:
151             def feat(e, ind_=ind, val_=val):
152                 return e[ind_] == val_
153             if self.header:
154                 feat.__doc__ = self.header[ind]+"==" +str(val)
155             else:
156                 feat.__doc__ = "e["+str(ind)+"("==" +str(val)
157             feat.frange = boolean
158             feat.ftype = "boolean"
159             conds.append(feat)
160 self.conditions_cache[(max_num_cuts, categorical_only)] = conds
161 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 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)
162 def evaluate_dataset(self, data, predictor, error_measure):
163     """Evaluates predictor on data according to the error_measure
164     predictor is a function that takes an example and returns a
165     prediction for the target features.
166     error_measure(prediction,actual) -> non-negative real
167     """
168     if data:
169         try:
170             value = statistics.mean(error_measure(predictor(e),
171                                                    self.target(e))
172                                   for e in data)
173         except ValueError: # if error_measure gives an error
174             return float("inf") # infinity
175         return value
176     else:
177         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)
178 class Evaluate(object):
179     """A container for the evaluation measures"""
180
181     def squared_loss(prediction, actual):
182         "squared loss "
183         if isinstance(prediction, (list,dict)):
184             return (1-prediction[actual])**2 # the correct value is 1
185         else:
186             return (prediction-actual)**2
187
188     def absolute_loss(prediction, actual):
189         "absolute loss "
190         if isinstance(prediction, (list,dict)):
191             return abs(1-prediction[actual]) # the correct value is 1
192         else:
193             return abs(prediction-actual)
194
195     def log_loss(prediction, actual):
196         "log loss (bits)"
197         try:
198             if isinstance(prediction, (list,dict)):
199                 return -math.log2(prediction[actual])
200             else:
201                 return -math.log2(prediction) if actual==1 else
202                     -math.log2(1-prediction)
203         except ValueError:
204             return float("inf") # infinity
205
206     def accuracy(prediction, actual):
207         "accuracy "
208         if isinstance(prediction, dict):
209             prev_val = prediction[actual]
210             return 1 if all(prev_val >= v for v in prediction.values())
211                 else 0
212         if isinstance(prediction, list):
213             prev_val = prediction[actual]
214             return 1 if all(prev_val >= v for v in prediction) else 0
215         else:
216             return 1 if abs(actual-prediction) <= 0.5 else 0
217
218     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)
218 def partition_data(data, prob_test=0.30):
219     """partitions the data into a training set and a test set, where
220     prob_test is the probability of each example being in the test set.
221     """
222     train = []
223     test = []
224     for example in data:
225         if random.random() < prob_test:
226             test.append(example)
227         else:
228             train.append(example)
229     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)
231 class Data_from_file(Data_set):
232     def __init__(self, file_name, separator=',', num_train=None,
233                 prob_test=0.3,
234                 has_header=False, target_index=0, boolean_features=True,
235                 categorical=[], target_type=None, include_only=None,
236                 seed=None): #seed=12345):

```

```

235 """create a dataset from a file
236 separator is the character that separates the attributes
237 num_train is a number specifying the first num_train tuples are
      training, or None
238 prob_test is the probability an example should in the test set (if
      num_train is None)
239 has_header is True if the first line of file is a header
240 target_index specifies which feature is the target
241 boolean_features specifies whether we want to create Boolean
      features
      (if False, it uses the original features).
242 categorical is a set (or list) of features that should be treated
      as categorical
243 target_type is either None for automatic detection of target type
244 or one of "numeric", "boolean", "cartegorical"
245 include_only is a list or set of indexes of columns to include
246 """
247
248 self.boolean_features = boolean_features
249 with open(file_name,'r',newline='') as csvfile:
250     self.display(1,"Loading",file_name)
251     # data_all = csv.reader(csvfile,delimiter=separator) # for more
      complicated CSV files
252     data_all = (line.strip().split(separator) for line in csvfile)
253     if include_only is not None:
254         data_all = ([v for (i,v) in enumerate(line) if i in
      include_only]
255                     for line in data_all)
256     if has_header:
257         header = next(data_all)
258     else:
259         header = None
260     data_tuples = (interpret_elements(d) for d in data_all if
      len(d)>1)
261     if num_train is not None:
262         # training set is divided into training then test examples
263         # the file is only read once, and the data is placed in
      appropriate list
264         train = []
265         for i in range(num_train): # will give an error if
      insufficient examples
266             train.append(next(data_tuples))
267         test = list(data_tuples)
268         Data_set.__init__(self,train, test=test,
      target_index=target_index,header=header)
269     else: # randomly assign training and test examples
270         Data_set.__init__(self,data_tuples, test=None,
      prob_test=prob_test,
271                           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 dif-

ferent files

```

learnProblem.py — (continued)
273 class Data_from_files(Data_set):
274     def __init__(self, train_file_name, test_file_name, separator=',',
275                 has_header=False, target_index=0, boolean_features=True,
276                 categorical=[], target_type= None, include_only=None):
277         """create a dataset from separate training and file
278         separator is the character that separates the attributes
279         num_train is a number specifying the first num_train tuples are
280             training, or None
281         prob_test is the probability an example should in the test set (if
282             num_train is None)
283         has_header is True if the first line of file is a header
284         target_index specifies which feature is the target
285         boolean_features specifies whether we want to create Boolean
286             features
287             (if False, it uses the original features).
288         categorical is a set (or list) of features that should be treated
289             as categorical
290         target_type is either None for automatic detection of target type
291             or one of "numeric", "boolean", "cartegorical"
292         include_only is a list or set of indexes of columns to include
293         """
294         self.boolean_features = boolean_features
295         with open(train_file_name,'r',newline='') as train_file:
296             with open(test_file_name,'r',newline='') as test_file:
297                 # data_all = csv.reader(csvfile,delimiter=separator) # for more
298                 # complicated CSV files
299                 train_data = (line.strip().split(separator) for line in
300                             train_file)
301                 test_data = (line.strip().split(separator) for line in
302                             test_file)
303                 if include_only is not None:
304                     train_data = ([v for (i,v) in enumerate(line) if i in
305                                 include_only]
306                                for line in train_data)
307                     test_data = ([v for (i,v) in enumerate(line) if i in
308                                 include_only]
309                                for line in test_data)
310                 if has_header: # this assumes the training file has a header
311                     and the test file doesn't
312                     header = next(train_data)
313                 else:
314                     header = None
315                 train_tuples = [interpret_elements(d) for d in train_data if
316                               len(d)>1]
317                 test_tuples = [interpret_elements(d) for d in test_data if
318                               len(d)>1]
319                 Data_set.__init__(self,train_tuples, test_tuples,
320                                 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)
310 def interpret_elements(str_list):
311     """make the elements of string list str_list numeric if possible.
312     Otherwise remove initial and trailing spaces.
313     """
314     res = []
315     for e in str_list:
316         try:
317             res.append(int(e))
318         except ValueError:
319             try:
320                 res.append(float(e))
321             except ValueError:
322                 se = e.strip()
323                 if se in ["True", "true", "TRUE"]:
324                     res.append(True)
325                 if se in ["False", "false", "FALSE"]:
326                     res.append(False)
327                 else:
328                     res.append(e.strip())
329     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 **kernels**; 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)
331 class Data_set_augmented(Data_set):
332     def __init__(self, dataset, unary_functions=[], binary_functions=[],
333                 include_orig=True):
334         """creates a dataset like dataset but with new features
335         unary_function is a list of unary feature constructors
336         binary_functions is a list of binary feature combiners.
337         include_orig specifies whether the original features should be
338         included

```

```

337         """
338         self.orig_dataset = dataset
339         self.unary_functions = unary_functions
340         self.binary_functions = binary_functions
341         self.include_orig = include_orig
342         self.target = dataset.target
343         Data_set.__init__(self, dataset.train, test=dataset.test,
344                           target_index = dataset.target_index)
345
346     def create_features(self):
347         if self.include_orig:
348             self.input_features = self.orig_dataset.input_features.copy()
349         else:
350             self.input_features = []
351         for u in self.unary_functions:
352             for f in self.orig_dataset.input_features:
353                 self.input_features.append(u(f))
354         for b in self.binary_functions:
355             for f1 in self.orig_dataset.input_features:
356                 for f2 in self.orig_dataset.input_features:
357                     if f1 != f2:
358                         self.input_features.append(b(f1,f2))

```

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

```

learnProblem.py — (continued)
360 def square(f):
361     """a unary feature constructor to construct the square of a feature
362     """
363     def sq(e):
364         return f(e)**2
365     sq.__doc__ = f.__doc__+"**2"
366     return sq
367
368 def power_feat(n):
369     """given n returns a unary feature constructor to construct the nth
370     power of a feature.
371     e.g., power_feat(2) is the same as square, defined above
372     """
373     def fn(f,n=n):
374         def pow(e,n=n):
375             return f(e)**n
376         pow.__doc__ = f.__doc__+"**"+str(n)
377         return pow
378     return fn
379
380 def prod_feat(f1,f2):
381     """a new feature that is the product of features f1 and f2
382     """
383     def feat(e):

```

```

383         return f1(e)*f2(e)
384     feat.__doc__ = f1.__doc__+"*" +f2.__doc__
385     return feat
386
387 def eq_feat(f1,f2):
388     """a new feature that is 1 if f1 and f2 give same value
389     """
390     def feat(e):
391         return 1 if f1(e)==f2(e) else 0
392     feat.__doc__ = f1.__doc__+"==" +f2.__doc__
393     return feat
394
395 def neq_feat(f1,f2):
396     """a new feature that is 1 if f1 and f2 give different values
397     """
398     def feat(e):
399         return 1 if f1(e)!=f2(e) else 0
400     feat.__doc__ = f1.__doc__+"!=" +f2.__doc__
401     return feat

```

Example:

```

learnProblem.py — (continued)
403 # from learnProblem import Data_set_augmented,prod_feat
404 # data = Data_from_file('data/holiday.csv', num_train=19, target_index=-1)
405 # data = Data_from_file('data/iris.data', prob_test=1/3, target_index=-1)
406 ## Data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0)
407 # dataplus = Data_set_augmented(data,[],[prod_feat])
408 # 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)
409 from display import Displayable
410
411 class Learner(Displayable):
412     def __init__(self, dataset):
413         raise NotImplementedError("Learner.__init__") # abstract method

```



```

414 |
415 |     def learn(self):
416 |         """returns a predictor, a function from a tuple to a value for the
         target feature
417 |         """
418 |         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 if the values are ratings in $\{1, 2, 3, 4, 5\}$, we can only predict one of these integers.
- a point prediction, where we are allowed to predict any value. For example, if the values of the feature are $\{0, 1\}$ we may be allowed to predict 0.3, 1, or even 1.7. For all of the criteria we can imagine, there is no point in predicting a value greater than 1 or less than zero (but that doesn't mean we can't), but it is often useful to predict a value between 0 and 1. If the values are ratings in $\{1, 2, 3, 4, 5\}$, we may want to predict 3.4.
- a probability distribution over the values of the feature. For each value v , we predict a non-negative number p_v , such that the sum over all predictions is 1.

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
11 from learnProblem import Evaluate
12 import math, random, collections, statistics
13 import utilities # argmax for (element,value) pairs
14
15 class Predict(object):
16     """The class of prediction methods for a list of values.
17     Please make the doc strings the same length, because they are used in
18     tables.
19     Note that we don't need self argument, as we are creating Predict
20     objects,
21     To use call Predict.laplace(data) etc."""
22
23     ### The following return a distribution over values (for classification)
24     def empirical(data, domain=[0,1], icount=0):
25         "empirical dist "
26         # returns a distribution over values
27         counts = {v:icount for v in domain}
28         for e in data:
29             counts[e] += 1
30         s = sum(counts.values())
31         return {k:v/s for (k,v) in counts.items()}
32
33     def bounded_empirical(data, domain=[0,1], bound=0.01):
34         "bounded empirical"
35         return {k:min(max(v,bound),1-bound) for (k,v) in
36                 Predict.empirical(data, domain).items()}
37
38     def laplace(data, domain=[0,1]):
39         "Laplace " # for categorical data
40         return Predict.empirical(data, domain, icount=1)
41
42     def cmode(data, domain=[0,1]):
43         "mode " # for categorical data
44         md = statistics.mode(data)
45         return {v: 1 if v==md else 0 for v in domain}
46
47     def cmedian(data, domain=[0,1]):
48         "median " # for categorical data
49         md = statistics.median_low(data) # always return one of the values
50         return {v: 1 if v==md else 0 for v in domain}
51
52     ### The following return a single prediction (for regression). domain
53     is ignored.
54
55     def mean(data, domain=[0,1]):
56         "mean "
57         # returns a real number
58         return statistics.mean(data)

```

```

56 def rmean(data, domain=[0,1], mean0=0, pseudo_count=1):
57     "regularized mean"
58     # returns a real number.
59     # mean0 is the mean to be used for 0 data points
60     # With mean0=0.5, pseudo_count=2, same as laplace for [0,1] data
61     # this works for enumerations as well as lists
62     sum = mean0 * pseudo_count
63     count = pseudo_count
64     for e in data:
65         sum += e
66         count += 1
67     return sum/count
68
69 def mode(data, domain=[0,1]):
70     "mode"
71     return statistics.mode(data)
72
73 def median(data, domain=[0,1]):
74     "median"
75     return statistics.median(data)
76
77 all = [empirical, mean, rmean, bounded_empirical, laplace, cmode, mode,
78        median, cmedian]
79
80 # The following suggests appropriate predictions as a function of the
81 # target type
82 select = {"boolean": [empirical, bounded_empirical, laplace, cmode,
83                       cmedian],
84          "categorical": [empirical, bounded_empirical, laplace, cmode,
85                          cmedian],
86          "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)
83 def test_no_inputs(error_measures = Evaluate.all_criteria,
84                   num_samples=10000, test_size=10 ):
85     for train_size in [1,2,3,4,5,10,20,100,1000]:

```

```

85     results = {predictor: {error_measure: 0 for error_measure in
86         error_measures}
87         for predictor in Predict.all}
88     for sample in range(num_samples):
89         prob = random.random()
90         training = [1 if random.random() < prob else 0 for i in
91             range(train_size)]
92         test = [1 if random.random() < prob else 0 for i in
93             range(test_size)]
94         for predictor in Predict.all:
95             prediction = predictor(training)
96             for error_measure in error_measures:
97                 results[predictor][error_measure] += sum(
98                     error_measure(prediction, actual) for actual in
99                     test)/test_size
100     print(f"For training size {train_size}:")
101     print("  Predictor\t", "\t".join(error_measure.__doc__ for
102         error_measure in
103         error_measures), sep="\t")
104     for predictor in Predict.all:
105         print(f"  {predictor.__doc__}",
106             "\t".join("{:.7f}".format(results[predictor][error_measure]/num_samples)
107                 for error_measure in
108                 error_measures), sep="\t")
109
110 if __name__ == "__main__":
111     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 pseudo-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
11 from learnProblem import Learner, Evaluate
12 from learnNoInputs import Predict
13 import math
14
15 class DT_learner(Learner):
16     def __init__(self,
17                 dataset,
18                 split_to_optimize=Evaluate.log_loss, # to minimize for at
19                 each split
20                 leaf_prediction=Predict.empirical, # what to use for value
21                 at leaves
22                 train=None, # used for cross validation
23                 max_num_cuts=8, # maximum number of conditions to split a
24                 numeric feature into
25                 gamma=1e-7, # minimum improvement needed to expand a node
26                 min_child_weight=10):
27         self.dataset = dataset
28         self.target = dataset.target
29         self.split_to_optimize = split_to_optimize
30         self.leaf_prediction = leaf_prediction
31         self.max_num_cuts = max_num_cuts
32         self.gamma = gamma
33         self.min_child_weight = min_child_weight
34         if train is None:
35             self.train = self.dataset.train
36         else:
37             self.train = train
38
39     def learn(self, max_num_cuts=8):
40         """learn a decision tree"""
41         return self.learn_tree(self.dataset.conditions(self.max_num_cuts),
42                               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)
40 def learn_tree(self, conditions, data_subset):
41     """returns a decision tree
42     conditions is a set of possible conditions
43     data_subset is a subset of the data used to build this (sub)tree
44
45     where a decision tree is a function that takes an example and
46     makes a prediction on the target feature
47     """
48     self.display(2,f"learn_tree with {len(conditions)} features and
49                  {len(data_subset)} examples")
50     split, partn = self.select_split(conditions, data_subset)
51     if split is None: # no split; return a point prediction
52         prediction = self.leaf_value(data_subset, self.target.frange)
53         self.display(2,f"leaf prediction for {len(data_subset)}
54                      examples is {prediction}")
55         def leaf_fun(e):
56             return prediction
57         leaf_fun.__doc__ = str(prediction)
58         leaf_fun.num_leaves = 1
59         return leaf_fun
60     else: # a split succeeded
61         false_examples, true_examples = partn
62         rem_features = [fe for fe in conditions if fe != split]
63         self.display(2,"Splitting on",split.__doc__,"with examples
64                      len(true_examples),":",len(false_examples))
65         true_tree = self.learn_tree(rem_features,true_examples)
66         false_tree = self.learn_tree(rem_features,false_examples)
67         def fun(e):
68             if split(e):
69                 return true_tree(e)
70             else:
71                 return false_tree(e)
72         #fun = lambda e: true_tree(e) if split(e) else false_tree(e)
73         fun.__doc__ = (f"(if {split.__doc__} then {true_tree.__doc__})")

```

```

72         f" else {false_tree.__doc__}")
73     fun.num_leaves = true_tree.num_leaves + false_tree.num_leaves
74     return fun

```

```

learnDT.py — (continued) —
76     def leaf_value(self, egs, domain):
77         return self.leaf_prediction((self.target(e) for e in egs), domain)
78
79     def select_split(self, conditions, data_subset):
80         """finds best feature to split on.
81
82         conditions is a non-empty list of features.
83         returns feature, partition
84         where feature is an input feature with the smallest error as
85             judged by split_to_optimize or
86             feature==None if there are no splits that improve the error
87         partition is a pair (false_examples, true_examples) if feature is
            not None
88         """
89         best_feat = None # best feature
90         # best_error = float("inf") # infinity - more than any error
91         best_error = self.sum_losses(data_subset) - self.gamma
92         self.display(3, "no split has
93             error=", best_error, "with", len(conditions), "conditions")
94         best_partition = None
95         for feat in conditions:
96             false_examples, true_examples = partition(data_subset, feat)
97             if
98                 min(len(false_examples), len(true_examples)) >= self.min_child_weight:
99                 err = (self.sum_losses(false_examples)
100                     + self.sum_losses(true_examples))
101                 self.display(3, "split on", feat.__doc__, "has error=", err,
102                     "splits
103                     into", len(true_examples), ":", len(false_examples), "gamma=", self.gamma)
104                 if err < best_error:
105                     best_feat = feat
106                     best_error = err
107                     best_partition = false_examples, true_examples
108                 self.display(2, "best split is on", best_feat.__doc__,
109                     "with err=", best_error)
110         return best_feat, best_partition
111
112     def sum_losses(self, data_subset):
113         """returns sum of losses for dataset (with no more splits)
114         There a single prediction for all leaves using leaf_prediction
115         It is evaluated using split_to_optimize
116         """
117         prediction = self.leaf_value(data_subset, self.target.frange)
118         error = sum(self.split_to_optimize(prediction, self.target(e))
119             for e in data_subset)

```

```

117         return error
118
119     def partition(data_subset, feature):
120         """partitions the data_subset by the feature"""
121         true_examples = []
122         false_examples = []
123         for example in data_subset:
124             if feature(example):
125                 true_examples.append(example)
126             else:
127                 false_examples.append(example)
128         return false_examples, true_examples

```

Test cases:

```

learnDT.py — (continued)
131 from learnProblem import Data_set, Data_from_file
132
133 def testDT(data, print_tree=True, selections = None, **tree_args):
134     """Prints errors and the trees for various evaluation criteria and ways
135     to select leaves.
136     """
137     if selections == None: # use selections suitable for target type
138         selections = Predict.select[data.target.ftype]
139     evaluation_criteria = Evaluate.all_criteria
140     print("Split Choice", "Leaf Choice\t", "#leaves", '\t'.join(ecrit.__doc__
141         for ecrit in
142         evaluation_criteria), sep="\t")
143
144     for crit in evaluation_criteria:
145         for leaf in selections:
146             tree = DT_learner(data, split_to_optimize=crit,
147                 leaf_prediction=leaf,
148                 **tree_args).learn()
149             print(crit.__doc__, leaf.__doc__, tree.num_leaves,
150                 "\t".join("{:.7f}".format(data.evaluate_dataset(data.test,
151                     tree, ecrit))
152                     for ecrit in evaluation_criteria), sep="\t")
153
154             if print_tree:
155                 print(tree.__doc__)
156
157 #DT_learner.max_display_level = 4
158 if __name__ == "__main__":
159     # Choose one of the data files
160     #data=Data_from_file('data/SPECT.csv', target_index=0);
161     print("SPECT.csv")
162     #data=Data_from_file('data/iris.data', target_index=-1);
163     print("iris.data")
164     data = Data_from_file('data/carbool.csv', target_index=-1, seed=123)
165     #data = Data_from_file('data/mail_reading.csv', target_index=-1);
166     print("mail_reading.csv")

```



```

158 | #data = Data_from_file('data/holiday.csv', num_train=19,
      |       target_index=-1); print("holiday.csv")
159 | 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 than this.
- (b) A depth-bound on the depth of the tree.
- (c) An improvement bound such that a split is only carried out if error with the split is better than the error without the split by at least the improvement bound.

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

Exercise 7.8 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_____
11 from learnProblem import Data_set, Data_from_file, Evaluate
12 from learnNoInputs import Predict
13 from learnDT import DT_learner
14 import matplotlib.pyplot as plt
15 import random
16
17 class K_fold_dataset(object):
18     def __init__(self, training_set, num_folds):
19         self.data = training_set.train.copy()
20         self.target = training_set.target
21         self.input_features = training_set.input_features
22         self.num_folds = num_folds
23         self.conditions = training_set.conditions
24
25         random.shuffle(self.data)
26         self.fold_boundaries = [(len(self.data)*i)//num_folds
27                                for i in range(0,num_folds+1)]
28
29     def fold(self, fold_num):
30         for i in range(self.fold_boundaries[fold_num],
31                        self.fold_boundaries[fold_num+1]):
32             yield self.data[i]
33
34     def fold_complement(self, fold_num):
35         for i in range(0,self.fold_boundaries[fold_num]):
36             yield self.data[i]
37         for i in range(self.fold_boundaries[fold_num+1],len(self.data)):
38             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) _____

```

```

40 def validation_error(self, learner, error_measure, **other_params):
41     error = 0
42     try:
43         for i in range(self.num_folds):
44             predictor = learner(self,
45                                 train=list(self.fold_complement(i)),
46                                 **other_params).learn()
47             error += sum( error_measure(predictor(e), self.target(e))
48                           for e in self.fold(i))
49     except ValueError:
50         return float("inf") #infinity
51     return error/len(self.data)

```

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

```

learnCrossValidation.py — (continued)
52 def plot_error(data, criterion=Evaluate.squared_loss,
53               leaf_prediction=Predict.empirical,
54               num_folds=5, maxx=None, xscale='linear'):
55     """Plots the error on the validation set and the test set
56     with respect to settings of the minimum number of examples.
57     xscale should be 'log' or 'linear'
58     """
59     plt.ion()
60     plt.xscale(xscale) # change between log and linear scale
61     plt.xlabel("min_child_weight")
62     plt.ylabel("average "+criterion.__doc__)
63     folded_data = K_fold_dataset(data, num_folds)
64     if maxx == None:
65         maxx = len(data.train)//2+1
66     errors = [] # validation errors
67     terrors = [] # test set errors
68     for mcw in range(1,maxx):
69         errors.append(folded_data.validation_error(DT_learner,criterion,leaf_prediction=leaf_prediction,
70                                                    min_child_weight=mcw))
71         tree = DT_learner(data, criterion, leaf_prediction=leaf_prediction,
72                           min_child_weight=mcw).learn()
73         terrors.append(data.evaluate_dataset(data.test,tree,criterion))
74     plt.plot(range(1,maxx), errors, ls='-',color='k',
75             label="validation for "+criterion.__doc__)
76     plt.plot(range(1,maxx), terrors, ls='--',color='k',
77             label="test set for "+criterion.__doc__)
78     plt.legend()
79     plt.draw()

```

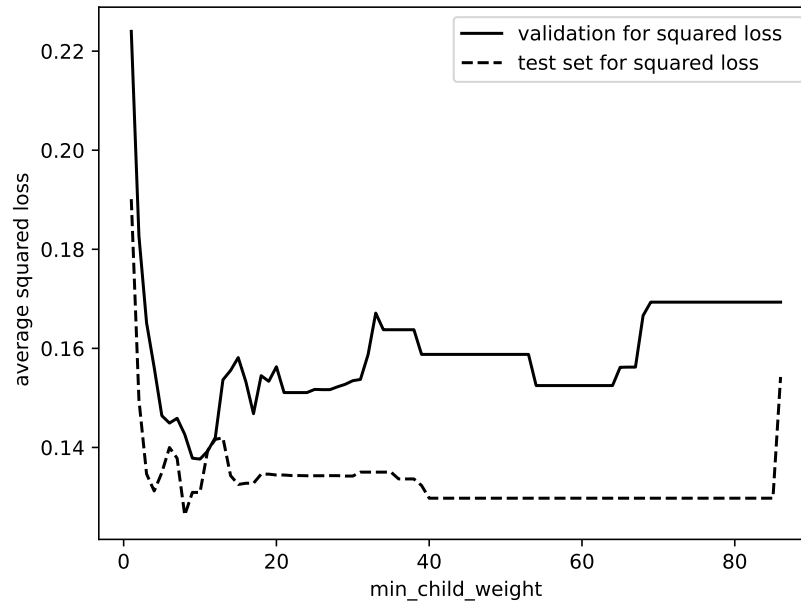


Figure 7.2: plot_error for SPECT dataset

```

79 # The following produces the graphs of Figure 7.18 of Poole and Mackworth
    [2023]
80 # data = Data_from_file('data/SPECT.csv', target_index=0, seed=123)
81 # plot_error(data, criterion=Evaluate.log_loss,
    leaf_prediction=Predict.laplace)
82
83 #also try:
84 # plot_error(data)
85 # 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 validation 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
11 from learnProblem import Learner
12 import random, math
13
14 class Linear_learner(Learner):
15     def __init__(self, dataset, train=None,
16                 learning_rate=0.1, max_init = 0.2,
17                 squashed=True, batch_size=10):
18         """Creates a gradient descent searcher for a linear classifier.
19         The main learning is carried out by learn()
20
21         dataset provides the target and the input features
22         train provides a subset of the training data to use
23         number_iterations is the default number of steps of gradient descent
24         learning_rate is the gradient descent step size
25         max_init is the maximum absolute value of the initial weights
26         squashed specifies whether the output is a squashed linear function
27         """
28         self.dataset = dataset
29         self.target = dataset.target
30         if train==None:
31             self.train = self.dataset.train
32         else:
33             self.train = train
34         self.learning_rate = learning_rate
35         self.squashed = squashed
36         self.batch_size = batch_size
37         self.input_features = [one]+dataset.input_features # one is defined
38             below
39         self.weights = {feat:random.uniform(-max_init,max_init)
40                         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)
41
42 def predictor(self,e):
43     """returns the prediction of the learner on example e"""
44     linpred = sum(w*f(e) for f,w in self.weights.items())
45     if self.squashed:
46         return sigmoid(linpred)
47     else:
48         return linpred
49
50 def predictor_string(self, sig_dig=3):

```

```

51     """returns the doc string for the current prediction function
52     sig_dig is the number of significant digits in the numbers"""
53     doc = "+".join(str(round(val,sig_dig))+"*"+feat.__doc__
54                     for feat,val in self.weights.items())
55     if self.squashed:
56         return "sigmoid("+ doc+")"
57     else:
58         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) _____
60     def learn(self,num_iter=100):
61         batch_size = min(self.batch_size, len(self.train))
62         d = {feat:0 for feat in self.weights}
63         for it in range(num_iter):
64             self.display(2,"prediction=",self.predictor_string())
65             for e in random.sample(self.train, batch_size):
66                 error = self.predictor(e) - self.target(e)
67                 update = self.learning_rate*error
68                 for feat in self.weights:
69                     d[feat] += update*feat(e)
70             for feat in self.weights:
71                 self.weights[feat] -= d[feat]
72                 d[feat]=0
73         return self.predictor

```

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

```

_____learnLinear.py — (continued) _____
75     def one(e):
76         "1"
77         return 1

```

sigmoid(*x*) is the function

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

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

```

_____learnLinear.py — (continued) _____
79     def sigmoid(x):
80         return 1/(1+math.exp(-x))
81
82     def logit(x):
83         return -math.log(1/x-1)

```

$\text{sigmoid}([x_0, v_2, \dots])$ 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)
85 def softmax(xs, domain=None):
86     """xs is a list of values, and
87     domain is the domain (a list) or None if the list should be returned
88     returns a distribution over the domain (a dict)
89     """
90     m = max(xs) # use of m prevents overflow (and all values underflowing)
91     exps = [math.exp(x-m) for x in xs]
92     s = sum(exps)
93     if domain:
94         return {d:v/s for (d,v) in zip(domain,exp)}
95     else:
96         return [v/s for v in exps]
97
98 def indicator(v, domain):
99     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)
101 from learnProblem import Data_set, Data_from_file, Evaluate
102 from learnProblem import Evaluate
103 import matplotlib.pyplot as plt
104
105 def test(**args):
106     data = Data_from_file('data/SPECT.csv', target_index=0)
107     # data = Data_from_file('data/mail_reading.csv', target_index=-1)
108     # data = Data_from_file('data/carbool.csv', target_index=-1)
109     learner = Linear_learner(data,**args)
110     learner.learn()
111     print("function learned is", learner.predictor_string())
112     for ecrit in Evaluate.all_criteria:
113         test_error = data.evaluate_dataset(data.test, learner.predictor,
114                                           ecrit)
115         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)
116 def plot_steps(learner=None,
117               data = None,
118               criterion=Evaluate.squared_loss,

```

```

119         step=1,
120         num_steps=1000,
121         log_scale=True,
122         legend_label=""):
123     """
124     plots the training and test error for a learner.
125     data is the
126     learner_class is the class of the learning algorithm
127     criterion gives the evaluation criterion plotted on the y-axis
128     step specifies how many steps are run for each point on the plot
129     num_steps is the number of points to plot
130
131     """
132     if legend_label != "": legend_label+=" "
133     plt.ion()
134     plt.xlabel("step")
135     plt.ylabel("Average "+criterion.__doc__)
136     if log_scale:
137         plt.xscale('log') #plt.semilogx() #Makes a log scale
138     else:
139         plt.xscale('linear')
140     if data is None:
141         data = Data_from_file('data/holiday.csv', num_train=19,
142                               target_index=-1)
143         #data = Data_from_file('data/SPECT.csv', target_index=0)
144         # data = Data_from_file('data/mail_reading.csv', target_index=-1)
145         # data = Data_from_file('data/carbool.csv', target_index=-1)
146     #random.seed(None) # reset seed
147     if learner is None:
148         learner = Linear_learner(data)
149     train_errors = []
150     test_errors = []
151     for i in range(1,num_steps+1,step):
152         test_errors.append(data.evaluate_dataset(data.test,
153                                                  learner.predictor, criterion))
154         train_errors.append(data.evaluate_dataset(data.train,
155                                                  learner.predictor, criterion))
156         learner.display(2, "Train error:",train_errors[-1],
157                        "Test error:",test_errors[-1])
158         learner.learn(num_iter=step)
159     plt.plot(range(1,num_steps+1,step),train_errors,ls='-',label=legend_label+"training")
160     plt.plot(range(1,num_steps+1,step),test_errors,ls='--',label=legend_label+"test")
161     plt.legend()
162     plt.draw()
163     learner.display(1, "Train error:",train_errors[-1],
164                    "Test error:",test_errors[-1])
165
166 if __name__ == "__main__":
167     test()

```

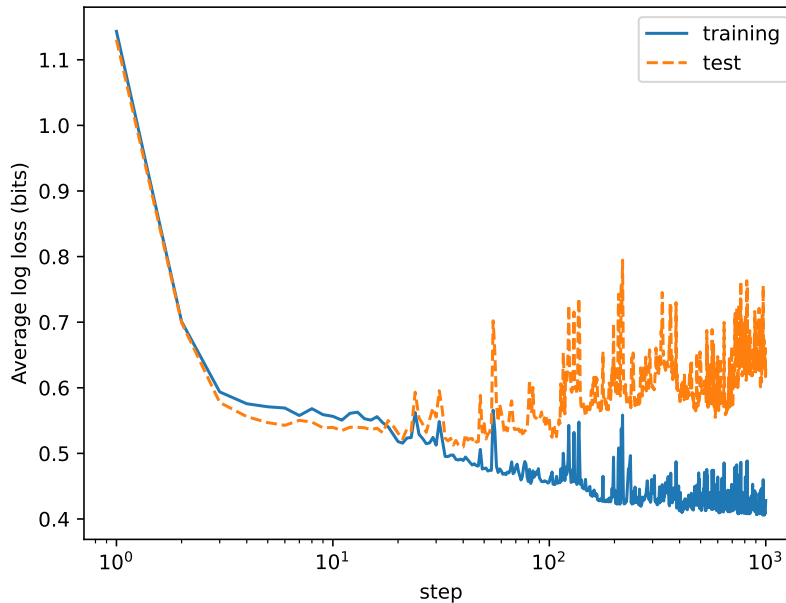



Figure 7.3: plot_steps for SPECT dataset

```

166 # This generates the figure
167 # from learnProblem import Data_set_augmented, prod_feat
168 # data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0,
    seed=123)
169 # dataplus = Data_set_augmented(data, [], [prod_feat])
170 # plot_steps(data=data, num_steps=1000)
171 # 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 underfitting 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)
172 def arange(start, stop, step):
173     """returns enumeration of values in the range [start, stop) separated by
    step.

```

```

174     like the built-in range(start,stop,step) but allows for integers and
        floats.
175     Note that rounding errors are expected with real numbers. (or use
        numpy.arange)
176     """
177     while start<stop:
178         yield start
179         start += step
180
181 def plot_prediction(data,
182                    learner = None,
183                    minx = 0,
184                    maxx = 5,
185                    step_size = 0.01, # for plotting
186                    label = "function"):
187     plt.ion()
188     plt.xlabel("x")
189     plt.ylabel("y")
190     if learner is None:
191         learner = Linear_learner(data, squashed=False)
192     learner.learning_rate=0.001
193     learner.learn(100)
194     learner.learning_rate=0.0001
195     learner.learn(1000)
196     learner.learning_rate=0.00001
197     learner.learn(10000)
198     learner.display(1,"function learned is", learner.predictor_string(),
199                   "error=",data.evaluate_dataset(data.train, learner.predictor,
200                                                  Evaluate.squared_loss))
201     plt.plot([e[0] for e in data.train],[e[-1] for e in
202           data.train],"bo",label="data")
203     plt.plot(list(arange(minx,maxx,step_size)),[learner.predictor([x])
204           for x in
205               arange(minx,maxx,step_size)],
206             label=label)
207
208     plt.legend()
209     plt.draw()

```

learnLinear.py — (continued)

```

207 from learnProblem import Data_set_augmented, power_feat
208 def plot_polynomials(data,
209                      learner_class = Linear_learner,
210                      max_degree = 5,
211                      minx = 0,
212                      maxx = 5,
213                      num_iter = 1000000,
214                      learning_rate = 0.00001,
215                      step_size = 0.01, # for plotting
216                      ):
217     plt.ion()

```

```

218 plt.xlabel("x")
219 plt.ylabel("y")
220 plt.plot([e[0] for e in data.train],[e[-1] for e in
      data.train],"ko",label="data")
221 x_values = list(arange(minx,maxx,step_size))
222 line_styles = ['-','--','-.',':']
223 colors = ['0.5','k','k','k','k']
224 for degree in range(max_degree):
225     data_aug = Data_set_augmented(data,[power_feat(n) for n in
      range(1,degree+1)],
226                                   include_orig=False)
227     learner = learner_class(data_aug,squashed=False)
228     learner.learning_rate = learning_rate
229     learner.learn(num_iter)
230     learner.display(1,"For degree",degree,
231                    "function learned is", learner.predictor_string(),
232                    "error=",data.evaluate_dataset(data.train,
      learner.predictor, Evaluate.squared_loss))
233     ls = line_styles[degree % len(line_styles)]
234     col = colors[degree % len(colors)]
235     plt.plot(x_values,[learner.predictor([x]) for x in x_values],
      linestyle=ls, color=col,
236              label="degree="+str(degree))
237     plt.legend(loc='upper left')
238     plt.draw()
239
240 # Try:
241 # data0 = Data_from_file('data/simp_regr.csv', prob_test=0,
      boolean_features=False, target_index=-1)
242 # plot_prediction(data0)
243 # plot_polynomials(data0)
244 # What if the step size was bigger?
245 # datam = Data_from_file('data/mail_reading.csv', target_index=-1)
246 # 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_____
11 from learnProblem import Data_set, Learner, Evaluate
12 from learnNoInputs import Predict
13 from learnLinear import sigmoid
14 import statistics
15 import random

```

```

16
17 class Boosted_dataset(Data_set):
18     def __init__(self, base_dataset, offset_fun, subsample=1.0):
19         """new dataset which is like base_dataset,
20         but offset_fun(e) is subtracted from the target of each example e
21         """
22         self.base_dataset = base_dataset
23         self.offset_fun = offset_fun
24         self.train =
25             random.sample(base_dataset.train, int(subsample*len(base_dataset.train)))
26         self.test = base_dataset.test
27         #Data_set.__init__(self, base_dataset.train, base_dataset.test,
28         #                    base_dataset.prob_test, base_dataset.target_index)
29
30     def create_features(self):
31         """creates new features - called at end of Data_set.init()
32         defines a new target
33         """
34         self.input_features = self.base_dataset.input_features
35         def newout(e):
36             return self.base_dataset.target(e) - self.offset_fun(e)
37         newout.frange = self.base_dataset.target.frange
38         newout.ftype = self.infer_type(newout.frange)
39         self.target = newout
40
41     def conditions(self, *args, colsample_bytree=0.5, **nargs):
42         conds = self.base_dataset.conditions(*args, **nargs)
43         return random.sample(conds, int(colsample_bytree*len(conds)))

```

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

learnBoosting.py — (continued)

```

44 class Boosting_learner(Learner):
45     def __init__(self, dataset, base_learner_class, subsample=0.8):
46         self.dataset = dataset
47         self.base_learner_class = base_learner_class
48         self.subsample = subsample
49         mean = sum(self.dataset.target(e)
50                 for e in self.dataset.train)/len(self.dataset.train)
51         self.predictor = lambda e:mean # function that returns mean for
52             each example
53         self.predictor.__doc__ = "lambda e:"+str(mean)
54         self.offsets = [self.predictor] # list of base learners
55         self.predictors = [self.predictor] # list of predictors
56         self.errors = [data.evaluate_dataset(data.test, self.predictor,
57             Evaluate.squared_loss)]
58         self.display(1,"Predict mean test set mean squared loss=",
59             self.errors[0] )

```

```

59     def learn(self, num_ensembles=10):
60         """adds num_ensemble learners to the ensemble.
61         returns a new predictor.
62         """
63         for i in range(num_ensembles):
64             train_subset = Boosted_dataset(self.dataset, self.predictor,
65                                           subsample=self.subsample)
66             learner = self.base_learner_class(train_subset)
67             new_offset = learner.learn()
68             self.offsets.append(new_offset)
69             def new_pred(e, old_pred=self.predictor, off=new_offset):
70                 return old_pred(e)+off(e)
71             self.predictor = new_pred
72             self.predictors.append(new_pred)
73             self.errors.append(data.evaluate_dataset(data.test,
74                                                    self.predictor, Evaluate.squared_loss))
75             self.display(1,f"Iteration {len(self.offsets)-1},treesize =
76                         {new_offset.num_leaves}. mean squared
77                         loss={self.errors[-1]}")
78         return self.predictor

```

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

learnBoosting.py — (continued)

```

76 # Testing
77
78 from learnDT import DT_learner
79 from learnProblem import Data_set, Data_from_file
80
81 def sp_DT_learner(split_to_optimize=Evaluate.squared_loss,
82                  leaf_prediction=Predict.mean,**nargs):
83     """Creates a learner with different default arguments replaced by
84     **nargs
85     """
86     def new_learner(dataset):
87         return DT_learner(dataset,split_to_optimize=split_to_optimize,
88                           leaf_prediction=leaf_prediction, **nargs)
89     return new_learner
90
91 #data = Data_from_file('data/car.csv', target_index=-1) regression
92 data = Data_from_file('data/student/student-mat-nq.csv',
93                      separator=';', has_header=True, target_index=-1, seed=13, include_only=list(range(30))+[32])
94 #2.0537973790924946
95 #data = Data_from_file('data/SPECT.csv', target_index=0, seed=62) #123)
96 #data = Data_from_file('data/mail_reading.csv', target_index=-1)
97 #data = Data_from_file('data/holiday.csv', num_train=19, target_index=-1)
98 #learner10 = Boosting_learner(data,
99                             sp_DT_learner(split_to_optimize=Evaluate.squared_loss,
100                            leaf_prediction=Predict.mean, min_child_weight=10))

```

```

96 #learner7 = Boosting_learner(data, sp_DT_learner(0.7))
97 #learner5 = Boosting_learner(data, sp_DT_learner(0.5))
98 #predictor9 = learner9.learn(10)
99 #for i in learner9.offsets: print(i.__doc__)
100 import matplotlib.pyplot as plt
101
102 def plot_boosting_trees(data, steps=10, mcws=[30,20,20,10], gammas=
    [100,200,300,500]):
103     # to reduce clutter uncomment one of following two lines
104     #mcws=[10]
105     #gammas=[200]
106     learners = [(mcw, gamma, Boosting_learner(data,
        sp_DT_learner(min_child_weight=mcw, gamma=gamma)))
107                 for gamma in gammas for mcw in mcws
108                 ]
109     plt.ion()
110     plt.xscale('linear') # change between log and linear scale
111     plt.xlabel("number of trees")
112     plt.ylabel("mean squared loss")
113     markers = (m+c for c in ['k','g','r','b','m','c','y'] for m in
        ['-','--','-.-',':'])
114     for (mcw,gamma,learner) in learners:
115         data.display(1,f"min_child_weight={mcw}, gamma={gamma}")
116         learner.learn(steps)
117         plt.plot(range(steps+1), learner.errors, next(markers),
118                 label=f"min_child_weight={mcw}, gamma={gamma}")
119     plt.legend()
120     plt.draw()
121
122 # 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)
124 class GTB_learner(DT_learner):
125     def __init__(self, dataset, number_trees, lambda_reg=1, gamma=0,
        **dtargs):

```

```

126         DT_learner.__init__(self, dataset,
127                             split_to_optimize=Evaluate.log_loss, **dtargs)
128         self.number_trees = number_trees
129         self.lambda_reg = lambda_reg
130         self.gamma = gamma
131         self.trees = []
132
133     def learn(self):
134         for i in range(self.number_trees):
135             tree =
136                 self.learn_tree(self.dataset.conditions(self.max_num_cuts),
137                                 self.train)
138             self.trees.append(tree)
139             self.display(1, f""""Iteration {i} treesize = {tree.num_leaves}
140                             train logloss={
141                                 self.dataset.evaluate_dataset(self.dataset.train,
142                                                             self.gtb_predictor, Evaluate.log_loss)
143                             } test logloss={
144                                 self.dataset.evaluate_dataset(self.dataset.test,
145                                                             self.gtb_predictor, Evaluate.log_loss)}""")
146         return self.gtb_predictor
147
148     def gtb_predictor(self, example, extra=0):
149         """prediction for example,
150         extras is an extra contribution for this example being considered
151         """
152         return sigmoid(sum(t(example) for t in self.trees)+extra)
153
154     def leaf_value(self, egs, domain=[0,1]):
155         """value at the leaves for examples egs
156         domain argument is ignored"""
157         predActs = [(self.gtb_predictor(e), self.target(e)) for e in egs]
158         return sum(a-p for (p,a) in predActs) / (sum(p*(1-p) for (p,a) in
159             predActs)+self.lambda_reg)
160
161     def sum_losses(self, data_subset):
162         """returns sum of losses for dataset (assuming a leaf is formed
163         with no more splits)
164         """
165         leaf_val = self.leaf_value(data_subset)
166         error = sum(Evaluate.log_loss(self.gtb_predictor(e, leaf_val),
167                                     self.target(e))
168                     for e in data_subset) + self.gamma
169         return error

```

Testing

```

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

```

```
165 | # gtb_learner.learn()
```


Neural Networks and Deep Learning

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

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
11 from learnProblem import Learner, Data_set, Data_from_file,
    Data_from_files, Evaluate
12 from learnLinear import sigmoid, one, softmax, indicator
13 import random, math, time
14
15 class Layer(object):
16     def __init__(self, nn, num_outputs=None):
17         """Given a list of inputs, outputs will produce a list of length
            num_outputs.
18         nn is the neural network this layer is part of
```

```

19         num outputs is the number of outputs for this layer.
20         """
21         self.nn = nn
22         self.num_inputs = nn.num_outputs # output of nn is the input to
           this layer
23         if num_outputs:
24             self.num_outputs = num_outputs
25         else:
26             self.num_outputs = nn.num_outputs # same as the inputs
27
28     def output_values(self, input_values, training=False):
29         """Return the outputs for this layer for the given input values.
30         input_values is a list of the inputs to this layer (of length
           num_inputs)
31         returns a list of length self.num_outputs.
32         It can act differently when training and when predicting.
33         """
34         raise NotImplementedError("output_values") # abstract method
35
36     def backprop(self, errors):
37         """Backpropagate the errors on the outputs
38         errors is a list of errors for the outputs (of length
           self.num_outputs).
39         Returns the errors for the inputs to this layer (of length
           self.num_inputs).
40
41         You can assume that this is only called after corresponding
           output_values,
42         which can remember information information required for the
           back-propagation.
43         """
44         raise NotImplementedError("backprop") # abstract method
45
46     def update(self):
47         """updates parameters after a batch.
48         overridden by layers that have parameters
49         """
50         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)

```

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

```

```

56     nn is a neural network that the inputs come from
57     num_outputs is the number of outputs
58     the random initialization of parameters is in range [-limit,limit]
59     """
60     Layer.__init__(self, nn, num_outputs)
61     if limit is None:
62         limit = math.sqrt(6/(self.num_inputs+self.num_outputs))
63     # self.weights[o][i] is the weight between input i and output o
64     self.weights = [[random.uniform(-limit, limit) if inf <
65                       self.num_inputs else 0
66                      for inf in range(self.num_inputs+1)]
67                     for outf in range(self.num_outputs)]
68     self.delta = [[0 for inf in range(self.num_inputs+1)]
69                   for outf in range(self.num_outputs)]
69
70     def output_values(self, input_values, training=False):
71         """Returns the outputs for the input values.
72         It remembers the values for the backprop.
73
74         Note in self.weights there is a weight list for every output,
75         so wts in self.weights loops over the outputs.
76         The bias is the *last* value of each list in self.weights.
77         """
78         self.inputs = input_values + [1]
79         return [sum(w*val for (w,val) in zip(wts,self.inputs))
80                for wts in self.weights]
81
82     def backprop(self, errors):
83         """Backpropagate the errors, updating the weights and returning the
84         error in its inputs.
85         """
86         input_errors = [0]*(self.num_inputs+1)
87         for out in range(self.num_outputs):
88             for inp in range(self.num_inputs+1):
89                 input_errors[inp] += self.weights[out][inp] * errors[out]
90                 self.delta[out][inp] += self.inputs[inp] * errors[out]
91         return input_errors[:-1] # remove the error for the "1"
92
93     def update(self):
94         """updates parameters after a batch"""
95         batch_step_size = self.nn.learning_rate / self.nn.batch_size
96         for out in range(self.num_outputs):
97             for inp in range(self.num_inputs+1):
98                 self.weights[out][inp] -= batch_step_size *
99                 self.delta[out][inp]
100                 self.delta[out][inp] = 0

```

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

learnNN.py — (continued)

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

```

101     """Rectified linear unit (ReLU)  $f(z) = \max(0, z)$ .
102     The number of outputs is equal to the number of inputs.
103     """
104     def __init__(self, nn):
105         Layer.__init__(self, nn)
106
107     def output_values(self, input_values, training=False):
108         """Returns the outputs for the input values.
109         It remembers the input values for the backprop.
110         """
111         self.input_values = input_values
112         self.outputs= [max(0,inp) for inp in input_values]
113         return self.outputs
114
115     def backprop(self,errors):
116         """Returns the derivative of the errors"""
117         return [e if inp>0 else 0 for e,inp in zip(errors,
118             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.

```

119 class Sigmoid_layer(Layer):
120     """sigmoids of the inputs.
121     The number of outputs is equal to the number of inputs.
122     Each output is the sigmoid of its corresponding input.
123     """
124     def __init__(self, nn):
125         Layer.__init__(self, nn)
126
127     def output_values(self, input_values, training=False):
128         """Returns the outputs for the input values.
129         It remembers the output values for the backprop.
130         """
131         self.outputs= [sigmoid(inp) for inp in input_values]
132         return self.outputs
133
134     def backprop(self,errors):
135         """Returns the derivative of the errors"""
136         return [e*out*(1-out) for e,out in zip(errors, self.outputs)]

```

8.2 Feedforward Networks

```

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

```

```

141         layers is the list of layers
142         """
143         self.dataset = dataset
144         self.output_type = dataset.target.ftype
145         self.learning_rate = learning_rate
146         self.input_features = dataset.input_features
147         self.num_outputs = len(self.input_features)
148         validation_num = int(len(self.dataset.train)*validation_proportion)
149         if validation_num > 0:
150             random.shuffle(self.dataset.train)
151             self.validation_set = self.dataset.train[-validation_num:]
152             self.training_set = self.dataset.train[:-validation_num]
153         else:
154             self.validation_set = []
155             self.training_set = self.dataset.train
156         self.layers = []
157         self.bn = 0 # number of batches run
158
159     def add_layer(self, layer):
160         """add a layer to the network.
161         Each layer gets number of inputs from the previous layers outputs.
162         """
163         self.layers.append(layer)
164         self.num_outputs = layer.num_outputs
165
166     def predictor(self, ex):
167         """Predicts the value of the first output for example ex.
168         """
169         values = [f(ex) for f in self.input_features]
170         for layer in self.layers:
171             values = layer.output_values(values)
172         return sigmoid(values[0]) if self.output_type == "boolean" \
173             else softmax(values, self.dataset.target.frange) if
174             self.output_type == "categorical" \
175             else values[0]
176
177     def predictor_string(self):
178         return "not implemented"

```

The *learn* method learns a network.

```

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

```

```

185     report_each means give the errors after each multiple of that
        iterations
186     """
187     self.batch_size = min(batch_size, len(self.training_set)) # don't
        have batches bigger than training size
188     if num_iter is None:
189         num_iter = (epochs * len(self.training_set)) // self.batch_size
190     #self.display(0, "Batch\t", "\t".join(criterion.__doc__ for criterion
        in Evaluate.all_criteria))
191     for i in range(num_iter):
192         batch = random.sample(self.training_set, self.batch_size)
193         for e in batch:
194             # compute all outputs
195             values = [f(e) for f in self.input_features]
196             for layer in self.layers:
197                 values = layer.output_values(values, training=True)
198             # backpropagate
199             predicted = [sigmoid(v) for v in values] if self.output_type
                == "boolean" \
200                 else softmax(values) if self.output_type ==
                "categorical" \
201                 else values
202             actuals = indicator(self.dataset.target(e),
                self.dataset.target.frange) \
203                 if self.output_type == "categorical" \
204                 else [self.dataset.target(e)]
205             errors = [pred-obsd for (obsd, pred) in
                zip(actuals, predicted)]
206             for layer in reversed(self.layers):
207                 errors = layer.backprop(errors)
208             # Update all parameters in batch
209             for layer in self.layers:
210                 layer.update()
211             self.bn+=1
212             if (i+1)%report_each==0:
213                 self.display(0, self.bn, "\t",
214                     "\t\t".join("{:.4f}".format(
215                         self.dataset.evaluate_dataset(self.validation_set,
                            self.predictor, criterion))
                            for criterion in Evaluate.all_criteria),
                            sep="")
216

```

8.3 Improved Optimization

8.3.1 Momentum

```

learnNN.py — (continued)
218 class Linear_complete_layer_momentum(Linear_complete_layer):
219     """a completely connected layer"""

```

```

220 def __init__(self, nn, num_outputs, limit=None, alpha=0.9, epsilon =
    1e-07, vel0=0):
221     """A completely connected linear layer.
222     nn is a neural network that the inputs come from
223     num_outputs is the number of outputs
224     max_init is the maximum value for random initialization of
        parameters
225     vel0 is the initial velocity for each parameter
226     """
227     Linear_complete_layer.__init__(self, nn, num_outputs, limit=limit)
228     # self.weights[o][i] is the weight between input i and output o
229     self.velocity = [[vel0 for inf in range(self.num_inputs+1)]
230                     for outf in range(self.num_outputs)]
231     self.alpha = alpha
232     self.epsilon = epsilon
233
234 def update(self):
235     """updates parameters after a batch"""
236     batch_step_size = self.nn.learning_rate / self.nn.batch_size
237     for out in range(self.num_outputs):
238         for inp in range(self.num_inputs+1):
239             self.velocity[out][inp] = self.alpha*self.velocity[out][inp]
                - batch_step_size * self.delta[out][inp]
240             self.weights[out][inp] += self.velocity[out][inp]
241             self.delta[out][inp] = 0

```

8.3.2 RMS-Prop

```

learnNN.py — (continued)
243 class Linear_complete_layer_RMS_Prop(Linear_complete_layer):
244     """a completely connected layer"""
245     def __init__(self, nn, num_outputs, limit=None, rho=0.9, epsilon =
        1e-07):
246         """A completely connected linear layer.
247         nn is a neural network that the inputs come from
248         num_outputs is the number of outputs
249         max_init is the maximum value for random initialization of
            parameters
250         """
251         Linear_complete_layer.__init__(self, nn, num_outputs, limit=limit)
252         # self.weights[o][i] is the weight between input i and output o
253         self.ms = [[0 for inf in range(self.num_inputs+1)]
254                 for outf in range(self.num_outputs)]
255         self.rho = rho
256         self.epsilon = epsilon
257
258     def update(self):
259         """updates parameters after a batch"""
260         for out in range(self.num_outputs):
261             for inp in range(self.num_inputs+1):

```

```

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

```

8.4 Dropout

Dropout is implemented as a layer.

```

learnNN.py — (continued)
267 from utilities import flip
268 class Dropout_layer(Layer):
269     """Dropout layer
270     """
271
272     def __init__(self, nn, rate=0):
273         """
274         rate is fraction of the input units to drop. 0 <= rate < 1
275         """
276         self.rate = rate
277         Layer.__init__(self, nn)
278
279     def output_values(self, input_values, training=False):
280         """Returns the outputs for the input values.
281         It remembers the input values for the backprop.
282         """
283         if training:
284             scaling = 1/(1-self.rate)
285             self.mask = [0 if flip(self.rate) else 1
286                          for _ in input_values]
287             return [x*y*scaling for (x,y) in zip(input_values, self.mask)]
288         else:
289             return input_values
290
291     def backprop(self, errors):
292         """Returns the derivative of the errors"""
293         return [x*y for (x,y) in zip(errors, self.mask)]
294
295 class Dropout_layer_0(Layer):
296     """Dropout layer
297     """
298
299     def __init__(self, nn, rate=0):
300         """
301         rate is fraction of the input units to drop. 0 <= rate < 1
302         """
303         self.rate = rate

```



```

304         Layer.__init__(self, nn)
305
306     def output_values(self, input_values, training=False):
307         """Returns the outputs for the input values.
308         It remembers the input values for the backprop.
309         """
310         if training:
311             scaling = 1/(1-self.rate)
312             self.outputs= [0 if flip(self.rate) else inp*scaling # make 0
313                             with probability rate
314                             for inp in input_values]
315             return self.outputs
316         else:
317             return input_values
318
319     def backprop(self,errors):
320         """Returns the derivative of the errors"""
321         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 categories (so it can use a softmax).

```

learnNN.py — (continued)
322 #data = Data_from_file('data/mail_reading.csv', target_index=-1)
323 #data = Data_from_file('data/mail_reading_consis.csv', target_index=-1)
324 data = Data_from_file('data/SPECT.csv', prob_test=0.3, target_index=0,
325                       seed=12345)
326 #data = Data_from_file('data/iris.data', prob_test=0.2, target_index=-1) #
327     150 examples approx 120 test + 30 test
328 #data = Data_from_file('data/if_x_then_y_else_z.csv', num_train=8,
329     target_index=-1) # not linearly sep
330 #data = Data_from_file('data/holiday.csv', target_index=-1) #,
331     num_train=19)
332 #data = Data_from_file('data/processed.cleveland.data', target_index=-1)
333 #random.seed(None)
334
335 # nn3 is has a single hidden layer of width 3
336 nn3 = NN(data, validation_proportion = 0)
337 nn3.add_layer(Linear_complete_layer(nn3,3))
338 #nn3.add_layer(Sigmoid_layer(nn3))
339 nn3.add_layer(ReLU_layer(nn3))
340 nn3.add_layer(Linear_complete_layer(nn3,1)) # when using
341     output_type="boolean"
342 #nn3.learn(epochs = 100)
343
344

```

```

339 # nn3do is like nn3 but with dropout on the hidden layer
340 nn3do = NN(data, validation_proportion = 0)
341 nn3do.add_layer(Linear_complete_layer(nn3do,3))
342 #nn3.add_layer(Sigmoid_layer(nn3)) # comment this or the next
343 nn3do.add_layer(ReLU_layer(nn3do))
344 nn3do.add_layer(Dropout_layer(nn3do, rate=0.5))
345 nn3do.add_layer(Linear_complete_layer(nn3do,1))
346 #nn3do.learn(epochs = 100)
347
348 # nn3_rmsp is like nn3 but uses RMS prop
349 nn3_rmsp = NN(data, validation_proportion = 0)
350 nn3_rmsp.add_layer(Linear_complete_layer_RMS_Prop(nn3_rmsp,3))
351 #nn3_rmsp.add_layer(Sigmoid_layer(nn3_rmsp)) # comment this or the next
352 nn3_rmsp.add_layer(ReLU_layer(nn3_rmsp))
353 nn3_rmsp.add_layer(Linear_complete_layer_RMS_Prop(nn3_rmsp,1))
354 #nn3_rmsp.learn(epochs = 100)
355
356 # nn3_m is like nn3 but uses momentum
357 mm1_m = NN(data, validation_proportion = 0)
358 mm1_m.add_layer(Linear_complete_layer_momentum(mm1_m,3))
359 #mm1_m.add_layer(Sigmoid_layer(mm1_m)) # comment this or the next
360 mm1_m.add_layer(ReLU_layer(mm1_m))
361 mm1_m.add_layer(Linear_complete_layer_momentum(mm1_m,1))
362 #mm1_m.learn(epochs = 100)
363
364 # nn2 has a single a hidden layer of width 2
365 nn2 = NN(data, validation_proportion = 0)
366 nn2.add_layer(Linear_complete_layer_RMS_Prop(nn2,2))
367 nn2.add_layer(ReLU_layer(nn2))
368 nn2.add_layer(Linear_complete_layer_RMS_Prop(nn2,1))
369
370 # nn5 is has a single hidden layer of width 5
371 nn5 = NN(data, validation_proportion = 0)
372 nn5.add_layer(Linear_complete_layer_RMS_Prop(nn5,5))
373 nn5.add_layer(ReLU_layer(nn5))
374 nn5.add_layer(Linear_complete_layer_RMS_Prop(nn5,1))
375
376 # nn0 has no hidden layers, and so is just logistic regression:
377 nn0 = NN(data, validation_proportion = 0) #learning_rate=0.05)
378 nn0.add_layer(Linear_complete_layer(nn0,1))
379 # Or try this for RMS-Prop:
380 #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)
382 from learnLinear import plot_steps
383 from learnProblem import Evaluate

```

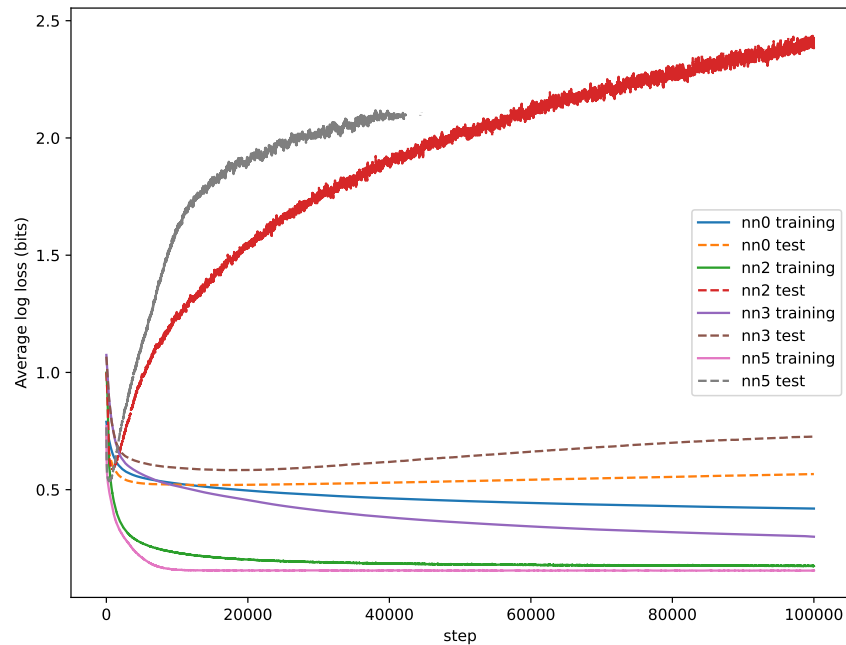


Figure 8.1: Plotting train and test log loss for various algorithms on SPECT dataset

```

384
385 # To show plots first choose a criterion to use
386 # crit = Evaluate.log_loss
387 # crit = Evaluate.accuracy
388 # plot_steps(learner = nn0, data = data, criterion=crit, num_steps=10000,
389             log_scale=False, legend_label="nn0")
389 # plot_steps(learner = nn2, data = data, criterion=crit, num_steps=10000,
390             log_scale=False, legend_label="nn2")
390 # plot_steps(learner = nn3, data = data, criterion=crit, num_steps=10000,
391             log_scale=False, legend_label="nn3")
391 # plot_steps(learner = nn5, data = data, criterion=crit, num_steps=10000,
392             log_scale=False, legend_label="nn5")
392
393 # for (nn,nname) in [(nn0,"nn0"),(nn2,"nn2"),(nn3,"nn3"),(nn5,"nn5")]:
394     plot_steps(learner = nn, data = data, criterion=crit,
395               num_steps=100000, log_scale=False, legend_label=nname)
394
395 # Print some training examples
396 #for eg in random.sample(data.train,10): print(eg,nn3.predictor(eg))
397
398 # Print some test examples
399 #for eg in random.sample(data.test,10): print(eg,nn3.predictor(eg))

```

```

400
401 # To see the weights learned in linear layers
402 # nn3.layers[0].weights
403 # nn3.layers[2].weights
404
405 # Print test:
406 # for e in data.train: print(e,nn0.predictor(e))
407
408 def test(data, hidden_widths = [5], epochs=100,
409         optimizers = [Linear_complete_layer,
410                        Linear_complete_layer_momentum,
411                        Linear_complete_layer_RMS_Prop]):
412     data.display(0,"Batch\t","\t".join(criterion.__doc__ for criterion in
413     Evaluate.all_criteria))
414     for optimizer in optimizers:
415         nn = NN(data)
416         for width in hidden_widths:
417             nn.add_layer(optimizer(nn,width))
418             nn.add_layer(ReLU_layer(nn))
419             if data.target.ftype == "boolean":
420                 nn.add_layer(optimizer(nn,1))
421             else:
422                 error(f"Not implemented: {data.output_type}")
423         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)
423 # Simplified version: (6000 training instances)
424 # data_mnist = Data_from_file('../MNIST/mnist_train.csv', prob_test=0.9,
425     target_index=0, boolean_features=False, target_type="categorical")
426
427 # Full version:
428 # data_mnist = Data_from_files('../MNIST/mnist_train.csv',
429     '../MNIST/mnist_test.csv', target_index=0, boolean_features=False,
430     target_type="categorical")
431
432 # nn_mnist = NN(data_mnist, validation_proportion = 0.02,
433     learning_rate=0.001) #validation set = 1200
434 # nn_mnist.add_layer(Linear_complete_layer_RMS_Prop(nn_mnist,512));
435     nn_mnist.add_layer(ReLU_layer(nn_mnist));
436     nn_mnist.add_layer(Linear_complete_layer_RMS_Prop(nn_mnist,10))
437 # start_time = time.perf_counter();nn_mnist.learn(epochs=1,
438     batch_size=128);end_time = time.perf_counter();print("Time:", end_time

```

```
    - start_time,"seconds") #1 epoch
432 # determine test error:
433 # data_mnist.evaluate_dataset(data_mnist.test, nn_mnist.predictor,
    Evaluate.accuracy)
434 # Print some random predictions:
435 # 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 *nn3* 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

Reasoning with Uncertainty

9.1 Representing Probabilistic Models

A probabilistic model uses the same definition of a variable as a CSP (Section 4.1.1, page 55). 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
11 from display import Displayable
12 import math
13
14 class Factor(Displayable):
15     nextid=0 # each factor has a unique identifier; for printing
16
17     def __init__(self,variables):
18         self.variables = variables # ordered list of variables
```

```

19     self.id = Factor.nextid
20     self.name = f"f{self.id}"
21     Factor.nextid += 1
22
23     def can_evaluate(self, assignment):
24         """True when the factor can be evaluated in the assignment
25         assignment is a {variable:value} dict
26         """
27         return all(v in assignment for v in self.variables)
28
29     def get_value(self, assignment):
30         """Returns the value of the factor given the assignment of values
31         to variables.
32         Needs to be defined for each subclass.
33         """
34         assert self.can_evaluate(assignment)
35         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)_____
36     def __str__(self):
37         """returns a string representing a summary of the factor"""
38         return f"{self.name}({'.'.join(str(var) for var in
39             self.variables)})"
40
41     def to_table(self, variables=None, given={}):
42         """returns a string representation of the factor.
43         Allows for an arbitrary variable ordering.
44         variables is a list of the variables in the factor
45         (can contain other variables)"""
46         if variables==None:
47             variables = [v for v in self.variables if v not in given]
48         else: #enforce ordering and allow for extra variables in ordering
49             variables = [v for v in variables if v in self.variables and v
50                 not in given]
51         head = "\t".join(str(v) for v in variables)
52         return head+"\n"+self.ass_to_str(variables, given, variables)
53
54     def ass_to_str(self, vars, asst, allvars):
55         #print(f"ass_to_str({vars}, {asst}, {allvars})")
56         if vars:
57             return "\n".join(self.ass_to_str(vars[1:], {**asst,
58                 vars[0]:val}, allvars)
59                 for val in vars[0].domain)
60         else:
61             return ("\t".join(str(asst[var]) for var in allvars)
62                 + "\t"+"{: .6f}".format(self.get_value(asst)) )

```



```
61 | __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)
63 class CPD(Factor):
64     def __init__(self, child, parents):
65         """represents P(variable | parents)
66         """
67         self.parents = parents
68         self.child = child
69         Factor.__init__(self, parents+[child])
70
71     def __str__(self):
72         """A brief description of a factor using in tracing"""
73         if self.parents:
74             return f"P({self.child}|{' '.join(str(p) for p in
75                 self.parents)})"
76         else:
77             return f"P({self.child})"
78     __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)
80 class ConstantCPD(CPD):
81     def __init__(self, variable, value):
82         CPD.__init__(self, variable, [])
83         self.value = value
84     def get_value(self, assignment):
85         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

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

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

```

probFactors.py — (continued)
87 from learnLinear import sigmoid, logit
88
89 class LogisticRegression(CPD):
90     def __init__(self, child, parents, weights):
91         """A logistic regression representation of a conditional
92            probability.
93            child is the Boolean (or 0/1) variable whose CPD is being defined
94            parents is the list of parents
95            weights is list of parameters, such that weights[i+1] is the weight
96            for parents[i]
97            """
98         assert len(weights) == 1+len(parents)
99         CPD.__init__(self, child, parents)
100        self.weights = weights
101
102    def get_value(self, assignment):
103        assert self.can_evaluate(assignment)
104        prob = sigmoid(self.weights[0]
105                      + sum(self.weights[i+1]*assignment[self.parents[i]]
106                          for i in range(len(self.parents))))
107        if assignment[self.child]: #child is true
108            return prob
109        else:
110            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 $0 \leq p_i \leq 1$. The semantics is defined as though there are $k+1$ hidden variables $Z_0, Z_1 \dots Z_k$, where $P(Z_0) = p_0$ and $P(Z_i | Y_i) = p_i$ for $i \geq 1$, 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)
110 class NoisyOR(CPD):
111     def __init__(self, child, parents, weights):
112         """A noisy representation of a conditional probability.
113            variable is the Boolean (or 0/1) child variable whose CPD is being
114            defined
115            parents is the list of Boolean (or 0/1) parents
116            weights is list of parameters, such that weights[i+1] is the weight
117            for parents[i]
118            """
119         assert len(weights) == 1+len(parents)
120         CPD.__init__(self, child, parents)
121        self.weights = weights

```

```

120
121     def get_value(self, assignment):
122         assert self.can_evaluate(assignment)
123         probfalse = (1-self.weights[0])*math.prod(1-self.weights[i+1]
124                                                     for i in
125                                                         range(len(self.parents))
126                                                         if
127                                                             assignment[self.parents[i]])
128
129         if assignment[self.child]:
130             return 1-probfalse
131         else:
132             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, \dots, V_k , the value of $f(V_1 = v_1, V_2 = v_2, \dots, V_k = v_k)$ is stored in $f[v_1][v_2] \dots [v_k]$.

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

```

_____probFactors.py — (continued)_____
131 from functools import reduce
132
133 class TabFactor(Factor):
134
135     def __init__(self, variables, values):
136         Factor.__init__(self, variables)
137         self.values = values
138
139     def get_value(self, assignment):
140         return self.get_val_rec(self.values, self.variables, assignment)
141
142     def get_val_rec(self, value, variables, assignment):
143         if variables == []:
144             return value
145         else:
146             return self.get_val_rec(value[assignment[variables[0]]],
147                                     variables[1:], assignment)

```

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

```

_____probFactors.py — (continued)_____
149 class Prob(CPD, TabFactor):

```

```

150 """A factor defined by a conditional probability table"""
151 def __init__(self, var, pars, cpt):
152     """Creates a factor from a conditional probability table, cpt
153     The cpt values are assumed to be for the ordering par+[var]
154     """
155     TabFactor.__init__(self, pars+[var], cpt)
156     self.child = var
157     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 _____
11 from display import Displayable
12 from probFactors import CPD
13 import matplotlib.pyplot as plt
14
15 class GraphicalModel(Displayable):
16     """The class of graphical models.
17     A graphical model consists of a title, a set of variables and a set of
18     factors.
19
20     vars is a set of variables
21     factors is a set of factors
22     """
23     def __init__(self, title, variables=None, factors=None):
24         self.title = title
25         self.variables = variables
26         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) _____
27 class BeliefNetwork(GraphicalModel):
28     """The class of belief networks."""
29
30     def __init__(self, title, variables, factors):
31         """vars is a set of variables
32         factors is a set of factors. All of the factors are instances of
33         CPD (e.g., Prob).
34         """

```

```

34     GraphicalModel.__init__(self, title, variables, factors)
35     assert all(isinstance(f,CPD) for f in factors)
36     self.var2cpt = {f.child:f for f in factors}
37     self.var2parents = {f.child:f.parents for f in factors}
38     self.children = {n:[] for n in self.variables}
39     for v in self.var2parents:
40         for par in self.var2parents[v]:
41             self.children[par].append(v)
42     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)_____
44 def topological_sort(self):
45     """creates a topological ordering of variables such that the
46     parents of
47     a node are before the node.
48     """
49     if self.topological_sort_saved:
50         return self.topological_sort_saved
51     next_vars = {n for n in self.var2parents if not self.var2parents[n]}
52     self.display(3,'topological_sort: next_vars',next_vars)
53     top_order=[]
54     while next_vars:
55         var = next_vars.pop()
56         self.display(3,'select variable',var)
57         top_order.append(var)
58         next_vars -= {ch for ch in self.children[var]
59                     if all(p in top_order for p in
60                         self.var2parents[ch])}
61         self.display(3,'var_with_no_parents_left',next_vars)
62     self.display(3,"top_order",top_order)
63     assert
64         set(top_order)==set(self.var2parents),(top_order,self.var2parents)
65     self.topologicalsort_saved=top_order
66     return top_order

```

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

```

_____probGraphicalModels.py — (continued)_____
65 def show(self):
66     plt.ion() # interactive
67     ax = plt.figure().gca()
68     ax.set_axis_off()
69     plt.title(self.title)
70     bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5")
71     for var in reversed(self.topological_sort()):

```

4-chain

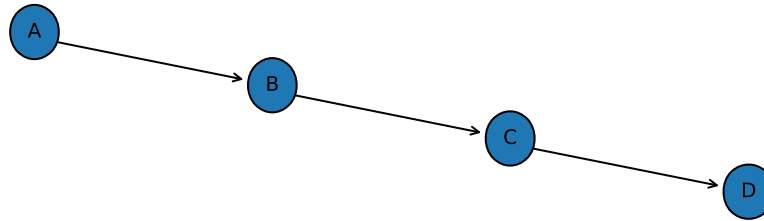


Figure 9.1: bn_4ch.show()

```

72         if self.var2parents[var]:
73             for par in self.var2parents[var]:
74                 ax.annotate(var.name, par.position, xytext=var.position,
75                             arrowprops={'arrowstyle': '<-'}, bbox=bbox,
76                             ha='center')
77         else:
78             x,y = var.position
79             plt.text(x,y,var.name,bbox=bbox,ha='center')

```

9.4.1 Example Belief Networks

A Chain of 4 Variables

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

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

```

_____probGraphicalModels.py — (continued) _____
81 from variable import Variable
82 from probFactors import Prob, LogisticRegression, NoisyOR
83
84 boolean = [False, True]
85 A = Variable("A", boolean, position=(0,0.8))
86 B = Variable("B", boolean, position=(0.333,0.7))
87 C = Variable("C", boolean, position=(0.666,0.6))
88 D = Variable("D", boolean, position=(1,0.5))
89
90 f_a = Prob(A,[],[0.4,0.6])
91 f_b = Prob(B,[A],[[0.9,0.1],[0.2,0.8]])
92 f_c = Prob(C,[B],[[0.6,0.4],[0.3,0.7]])
93 f_d = Prob(D,[C],[[0.1,0.9],[0.75,0.25]])
94
95 bn_4ch = BeliefNetwork("4-chain", {A,B,C,D}, {f_a,f_b,f_c,f_d})

```

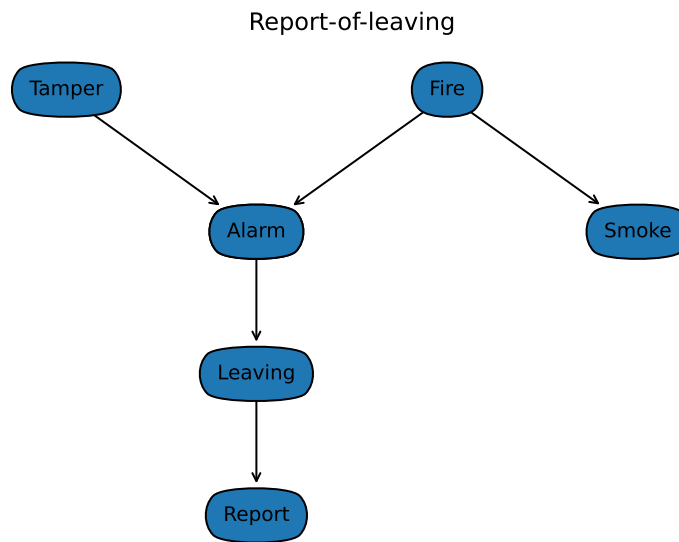


Figure 9.2: The report-of-leaving belief network

Report-of-Leaving Example

The second belief network, `bn_report`, is Example 8.15 of Poole and Mackworth [2017] (<http://artint.info>). The output of `bn_report.show()` is shown in Figure 9.2 of this document.

```

probGraphicalModels.py — (continued)
97 # Belief network report-of-leaving example (Example 8.15 shown in Figure
    8.3) of
98 # Poole and Mackworth, Artificial Intelligence, 2017 http://artint.info
99
100 Alarm = Variable("Alarm", boolean, position=(0.366,0.5))
101 Fire = Variable("Fire", boolean, position=(0.633,0.75))
102 Leaving = Variable("Leaving", boolean, position=(0.366,0.25))
103 Report = Variable("Report", boolean, position=(0.366,0.0))
104 Smoke = Variable("Smoke", boolean, position=(0.9,0.5))
105 Tamper = Variable("Tamper", boolean, position=(0.1,0.75))
106
107 f_ta = Prob(Tamper,[],[0.98,0.02])
108 f-fi = Prob(Fire,[],[0.99,0.01])
109 f-sm = Prob(Smoke,[Fire],[[0.99,0.01],[0.1,0.9]])
110 f-al = Prob(Alarm,[Fire,Tamper],[[0.9999, 0.0001], [0.15, 0.85]], [[0.01,
    0.99], [0.5, 0.5]])
111 f-lv = Prob(Leaving,[Alarm],[[0.999, 0.001], [0.12, 0.88]])
112 f-re = Prob(Report,[Leaving],[[0.99, 0.01], [0.25, 0.75]])

```

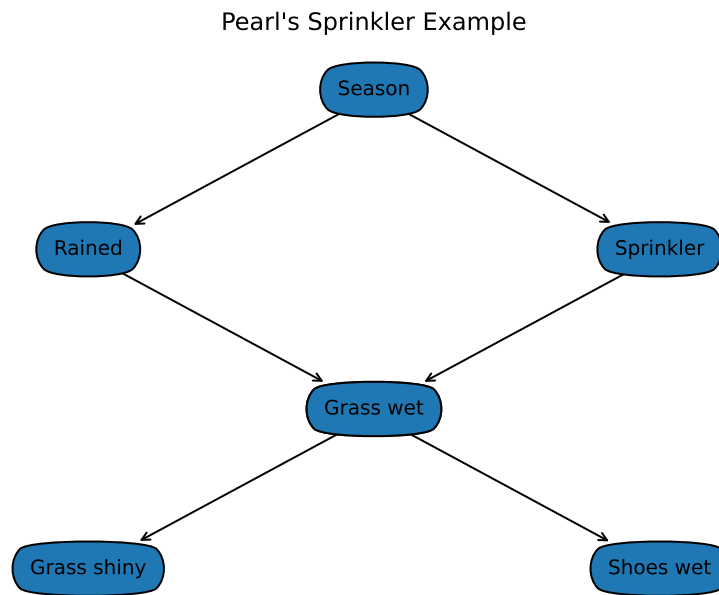


Figure 9.3: The sprinkler belief network

```

113 |
114 | bn_report = BeliefNetwork("Report-of-leaving",
115 |     {Tamper,Fire,Smoke,Alarm,Leaving,Report},
115 |     {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) _____
117 | Season = Variable("Season", ["summer","winter"], position=(0.5,0.9))
118 | Sprinkler = Variable("Sprinkler", ["on","off"], position=(0.9,0.6))
119 | Rained = Variable("Rained", boolean, position=(0.1,0.6))
120 | Grass_wet = Variable("Grass wet", boolean, position=(0.5,0.3))
121 | Grass_shiny = Variable("Grass shiny", boolean, position=(0.1,0))
122 | Shoes_wet = Variable("Shoes wet", boolean, position=(0.9,0))
123 |
124 | f_season = Prob(Season,[],{'summer':0.5, 'winter':0.5})
125 | f_sprinkler = Prob(Sprinkler,[Season],{'summer':{'on':0.9,'off':0.1},
126 |     'winter':{'on':0.01,'off':0.99}})
127 | f_rained = Prob(Rained,[Season],{'summer':[0.9,0.1], 'winter': [0.2,0.8]})
128 | f_wet = Prob(Grass_wet,[Sprinkler,Rained], {'on': [[0.1,0.9],[0.01,0.99]],
129 |     'off': [[0.99,0.01],[0.3,0.7]]})

```



```

130 f_shiny = Prob(Grass_shiny, [Grass_wet], [[0.95,0.05], [0.3,0.7]])
131 f_shoes = Prob(Shoes_wet, [Grass_wet], [[0.98,0.02], [0.35,0.65]])
132
133 bn_sprinkler = BeliefNetwork("Pearl's Sprinkler Example",
134                             {Season, Sprinkler, Rained, Grass_wet, Grass_shiny,
135                              Shoes_wet},
136                             {f_season, f_sprinkler, f_rained, f_wet, f_shiny,
137                              f_shoes})
138
139 bn_sprinkler_soff = BeliefNetwork("Pearl's Sprinkler Example
140                                  (do(Sprinkler=off))",
141                                  {Season, Sprinkler, Rained, Grass_wet, Grass_shiny,
142                                   Shoes_wet},
143                                  {f_season, f_rained, f_wet, f_shiny, f_shoes,
144                                   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 symptoms depend on the diseases, where the CPDs are defined using noisy-or. Bipartite means it is in two parts; the diseases are only connected to the symptoms and the symptoms are only connected to the diseases. The output of `bn_no1.show()` is shown in Figure 9.4 of this document.

```

_____probGraphicalModels.py — (continued)_____
142 Cough = Variable("Cough", boolean, (0.1,0.1))
143 Fever = Variable("Fever", boolean, (0.5,0.1))
144 Sneeze = Variable("Sneeze", boolean, (0.9,0.1))
145 Cold = Variable("Cold",boolean, (0.1,0.9))
146 Flu = Variable("Flu",boolean, (0.5,0.9))
147 Covid = Variable("Covid",boolean, (0.9,0.9))
148
149 p_cold_no = Prob(Cold,[],[0.9,0.1])
150 p_flu_no = Prob(Flu,[],[0.95,0.05])
151 p_covid_no = Prob(Covid,[],[0.99,0.01])
152
153 p_cough_no = NoisyOR(Cough, [Cold,Flu,Covid], [0.1, 0.3, 0.2, 0.7])
154 p_fever_no = NoisyOR(Fever, [ Flu,Covid], [0.01, 0.6, 0.7])
155 p_sneeze_no = NoisyOR(Sneeze, [Cold,Flu ], [0.05, 0.5, 0.2 ])
156
157 bn_no1 = BeliefNetwork("Bipartite Diagnostic Network (noisy-or)",
158                        {Cough, Fever, Sneeze, Cold, Flu, Covid},
159                        {p_cold_no, p_flu_no, p_covid_no, p_cough_no,
160                         p_fever_no, p_sneeze_no})
160
161 # to see the conditional probability of Noisy-or do:
162 # print(p_cough_no.to_table())
163

```

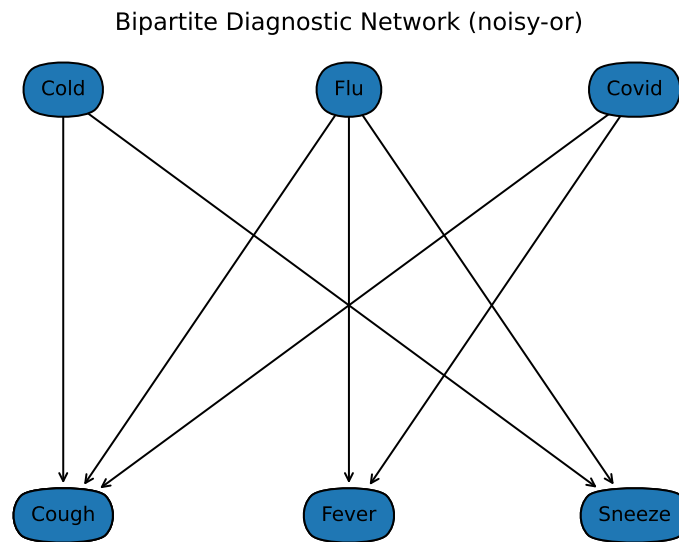


Figure 9.4: A bipartite diagnostic network

```

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

```

Bipartite Diagnostic Model with Logistic Regression

The belief network `bn_lr1` is a bipartite diagnostic model, with independent diseases, and the symptoms depend on the diseases, where the CPDs are defined using logistic regression. It has the same graphical structure as the previous example (see Figure 9.4). This has the (approximately) the same conditional probabilities as the previous example when zero or one diseases are present. Note that $\text{sigmoid}(-2.2) \approx 0.1$

```

_____probGraphicalModels.py — (continued) _____
169
170 p_cold_lr = Prob(Cold,[],[0.9,0.1])
171 p_flu_lr = Prob(Flu,[],[0.95,0.05])
172 p_covid_lr = Prob(Covid,[],[0.99,0.01])
173
174 p_cough_lr = LogisticRegression(Cough, [Cold,Flu,Covid], [-2.2, 1.67,
    1.26, 3.19])

```

```

175 p_fever_lr = LogisticRegression(Fever, [ Flu,Covid], [-4.6,      5.02,
      5.46])
176 p_sneeze_lr = LogisticRegression(Sneeze, [Cold,Flu ], [-2.94, 3.04, 1.79
      ])
177
178 bn_lr1 = BeliefNetwork("Bipartite Diagnostic Network - logistic
      regression",
179                        {Cough, Fever, Sneeze, Cold, Flu, Covid},
180                        {p_cold_lr, p_flu_lr, p_covid_lr, p_cough_lr,
                          p_fever_lr, p_sneeze_lr})
181
182 # to see the conditional probability of Noisy-or do:
183 #print(p_cough_lr.to_table())
184
185 # example from box "Noisy-or compared to logistic regression"
186 # from learnLinear import sigmoid, logit
187 # w0=logit(0.01)
188 # X = Variable("X",boolean)
189 # 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}))
190 # try to predict what would happen (and then test) if we had
191 # 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) _____
193 from display import Displayable
194
195 class InferenceMethod(Displayable):
196     """The abstract class of graphical model inference methods"""
197     method_name = "unnamed" # each method should have a method name
198
199     def __init__(self,gm=None):
200         self.gm = gm
201
202     def query(self, qvar, obs={}):
203         """returns a {value:prob} dictionary for the query variable"""
204         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) _____

```

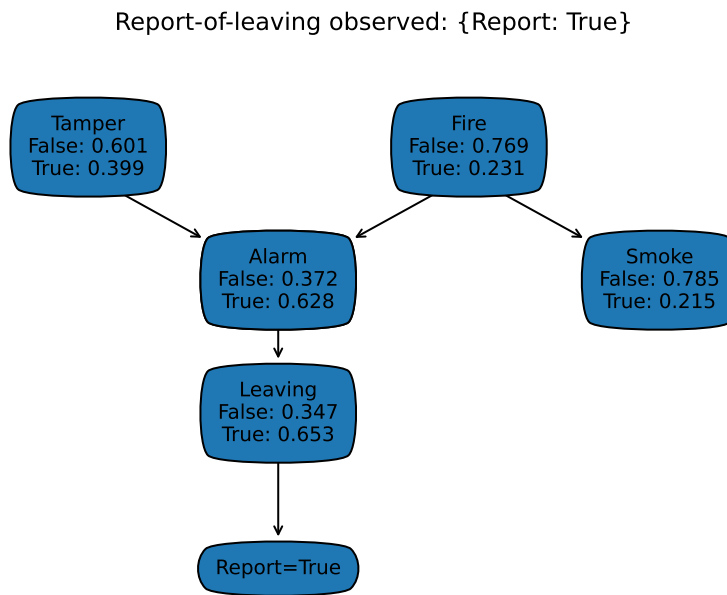


Figure 9.5: The report-of-leaving belief network with posterior distributions

```

206 def testIM(self, threshold=0.000000001):
207     solver = self.bn_4ch
208     res = solver.query(B,{D:True})
209     correct_answer = 0.429632380245
210     assert correct_answer-threshold < res[True] <
211           correct_answer+threshold, \
212           f"value {res[True]} not in desired range for
213           {self.method_name}"
212     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)
214 def show_post(self, obs={}, format_string="{:.3f}"):
215     """draws the graphical model conditioned on observations obs
216     format_string is changed if you want more or less precision
217     """
218     plt.ion() # interactive
219     ax = plt.figure().gca()
220     ax.set_axis_off()
221     plt.title(self.gm.title+" observed: "+str(obs))
222     bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5")
223     for var in reversed(self.gm.topological_sort()):

```

```

224         distn = self.query(var, obs=obs)
225         if var in obs:
226             text = var.name + "=" + str(obs[var])
227         else:
228             text = var.name + "\n" + "\n".join(str(d)+":
229             "+format_string.format(v) for (d,v) in distn.items())
230         if self.gm.var2parents[var]:
231             for par in self.gm.var2parents[var]:
232                 ax.annotate(text, par.position, xytext=var.position,
233                             arrowprops={'arrowstyle':'<-'},bbox=bbox,
234                                     ha='center')
235         else:
236             x,y = var.position
237             plt.text(x,y,text,bbox=bbox,ha='center')

```

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_____
11 import math
12 from probGraphicalModels import GraphicalModel, InferenceMethod
13 from probFactors import Factor
14
15 class ProbSearch(InferenceMethod):
16     """The class that queries graphical models using recursive conditioning
17
18     gm is graphical model to query
19     """
20     method_name = "recursive conditioning"
21
22     def __init__(self, gm=None):
23         InferenceMethod.__init__(self, gm)
24         ## self.max_display_level = 3
25
26     def query(self, qvar, obs={}, split_order=None):
27         """computes P(qvar | obs) where
28         qvar is the query variable
29         obs is a variable:value dictionary
30         split_order is a list of the non-observed non-query variables in gm
31         """
32         if qvar in obs:
33             return {val:(1 if val == obs[qvar] else 0) for val in
34                     qvar.domain}
35         else:
36             if split_order == None:

```

```

36         split_order = [v for v in self.gm.variables if (v not in
37                        obs) and v != qvar]
38         unnorm = [self.prob_search({qvar:val}|obs, self.gm.factors,
39                        split_order)
40                        for val in qvar.domain]
41         p_obs = sum(unnorm)
42         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)
42 def prob_search(self, context, factors, split_order):
43     """simple search algorithm
44     context is a variable:value dictionary
45     factors is a set of factors
46     split_order is a list of variables in factors not assigned in
47         context
48     returns sum over variable assignments to variables in split order
49         or product of factors """
50     self.display(2,"calling prob_search,", (context,factors))
51     if not factors:
52         return 1
53     elif to_eval := {fac for fac in factors if
54                     fac.can_evaluate(context)}:
55         # evaluate factors when all variables are assigned
56         self.display(3,"prob_search evaluating factors",to_eval)
57         val = math.prod(fac.get_value(context) for fac in to_eval)
58         return val * self.prob_search(context, factors-to_eval,
59                                     split_order)
60     else:
61         total = 0
62         var = split_order[0]
63         self.display(3, "prob_search branching on", var)
64         for val in var.domain:
65             total += self.prob_search({var:val}|context, factors,
66                                     split_order[1:])
67         self.display(3, "prob_search branching on", var,"returning",
68                     total)
69         return total

```

9.7 Recursive Conditioning

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

probRC.py — (continued)

```

65 class ProbRC(ProbSearch):
66     def __init__(self, gm=None):
67         self.cache = {(frozenset(), frozenset()):1}
68         ProbSearch.__init__(self, gm)
69
70     def prob_search(self, context, factors, split_order):
71         """ returns the number \sum_{split_order} \prod_{factors} given
72             assignments in context
73         context is a variable:value dictionary
74         factors is a set of factors
75         split_order is a list of variables in factors that are not assigned
76             in context
77         returns sum over variable assignments to variables in split_order
78             of the product of factors
79         """
80         self.display(3, "calling rc,", (context, factors))
81         ce = (frozenset(context.items()), frozenset(factors)) # key for the
82             cache entry
83         if ce in self.cache:
84             self.display(3, "rc cache lookup", (context, factors))
85             return self.cache[ce]
86         # if not factors: # no factors; needed if you don't have forgetting
87         # and caching
88         # return 1
89         elif vars_not_in_factors := {var for var in context
90             if not any(var in fac.variables for
91                 fac in factors)}:
92             # forget variables not in any factor
93             self.display(3, "rc forgetting variables", vars_not_in_factors)
94             return self.prob_search({key:val for (key,val) in
95                 context.items()
96                 if key not in vars_not_in_factors},
97                 factors, split_order)
98         elif to_eval := {fac for fac in factors if
99             fac.can_evaluate(context)}:
100             # evaluate factors when all variables are assigned
101             self.display(3, "rc evaluating factors", to_eval)
102             val = math.prod(fac.get_value(context) for fac in to_eval)
103             if val == 0:
104                 return 0
105             else:
106                 return val * self.prob_search(context, {fac for fac in factors
107                     if fac not in to_eval},
108                     split_order)
109         elif len(comp := connected_components(context, factors,
110             split_order)) > 1:
111             # there are disconnected components
112             self.display(3, "splitting into connected components", comp, "in
113                 context", context)

```

```

104         return(math.prod(self.prob_search(context,f,eo) for (f,eo) in
105                               comp))
106     else:
107         assert split_order, "split_order should not be empty to get
108                               here"
109         total = 0
110         var = split_order[0]
111         self.display(3, "rc branching on", var)
112         for val in var.domain:
113             total += self.prob_search({var:val}|context, factors,
114                                       split_order[1:])
115         self.cache[ce] = total
116         self.display(2, "rc branching on", var,"returning", total)
117     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)
116 def connected_components(context, factors, split_order):
117     """returns a list of (f,e) where f is a subset of factors and e is a
118       subset of split_order
119       such that each element shares the same variables that are disjoint from
120       other elements.
121     """
122     other_factors = set(factors) #copies factors
123     factors_to_check = {other_factors.pop()} # factors in connected
124       component still to be checked
125     component_factors = set() # factors in first connected component
126       already checked
127     component_variables = set() # variables in first connected component
128     while factors_to_check:
129         next_fac = factors_to_check.pop()
130         component_factors.add(next_fac)
131         new_vars = set(next_fac.variables) - component_variables -
132           context.keys()
133         component_variables |= new_vars

```



```

129         for var in new_vars:
130             factors_to_check |= {f for f in other_factors if var in
131                                 f.variables}
132             other_factors -= factors_to_check # set difference
133         if other_factors:
134             return ( [(component_factors,[e for e in split_order if e in
135                             component_variables])]
136                     + connected_components(context, other_factors, [e for e in
137                             split_order
138                                     if e not in
139                                     component_variables])
140                     )
141         else:
142             return [(component_factors, split_order)]

```

Testing:

```

probRC.py — (continued)
139 from probGraphicalModels import bn_4ch, A,B,C,D,f_a,f_b,f_c,f_d
140 bn_4chv = ProbRC(bn_4ch)
141 ## bn_4chv.query(A,{})
142 ## bn_4chv.query(D,{})
143 ## InferenceMethod.max_display_level = 3 # show more detail in displaying
144 ## InferenceMethod.max_display_level = 1 # show less detail in displaying
145 ## bn_4chv.query(A,{D:True},[C,B])
146 ## bn_4chv.query(B,{A:True,D:False})
147
148 from probGraphicalModels import
149     bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
150 bn_reportRC = ProbRC(bn_report) # answers queries using recursive
151     conditioning
152 ## bn_reportRC.query(Tamper,{})
153 ## InferenceMethod.max_display_level = 0 # show no detail in displaying
154 ## bn_reportRC.query(Leaving,{})
155 ## bn_reportRC.query(Tamper,{},
156     split_order=[Smoke,Fire,Alarm,Leaving,Report])
157 ## bn_reportRC.query(Tamper,{Report:True})
158 ## bn_reportRC.query(Tamper,{Report:True,Smoke:False})
159 ## Note what happens to the cache when these are called in turn:
160 ## bn_reportRC.query(Tamper,{Report:True},
161     split_order=[Smoke,Fire,Alarm,Leaving])
162 ## bn_reportRC.query(Smoke,{Report:True},
163     split_order=[Tamper,Fire,Alarm,Leaving])
164
165 from probGraphicalModels import bn_sprinkler, Season, Sprinkler, Rained,
166     Grass_wet, Grass_shiny, Shoes_wet
167 bn_sprinklerv = ProbRC(bn_sprinkler)
168 ## bn_sprinklerv.query(Shoes_wet,{})
169 ## bn_sprinklerv.query(Shoes_wet,{Rained:True})
170 ## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:True})
171 ## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:False,Rained:True})

```

```

166
167 from probGraphicalModels import bn_no1, bn_lr1, Cough, Fever, Sneeze,
    Cold, Flu, Covid
168 bn_no1v = ProbRC(bn_no1)
169 bn_lr1v = ProbRC(bn_lr1)
170 ## bn_no1v.query(Flu, {Fever:1, Sneeze:1})
171 ## bn_lr1v.query(Flu, {Fever:1, Sneeze:1})
172 ## bn_lr1v.query(Cough,{})
173 ## bn_lr1v.query(Cold,{Cough:1,Sneeze:0,Fever:1})
174 ## bn_lr1v.query(Flu,{Cough:0,Sneeze:1,Fever:1})
175 ## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1})
176 ## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:0})
177 ## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:1})
178
179 if __name__ == "__main__":
180     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_____
11 from probFactors import Factor, FactorObserved, FactorSum, factor_times
12 from probGraphicalModels import GraphicalModel, InferenceMethod
13
14 class VE(InferenceMethod):
15     """The class that queries Graphical Models using variable elimination.
16
17     gm is graphical model to query
18     """
19     method_name = "variable elimination"
20
21     def __init__(self, gm=None):
22         InferenceMethod.__init__(self, gm)
23
24     def query(self, var, obs={}, elim_order=None):
25         """computes P(var|obs) where
26         var is a variable
27         obs is a {variable:value} dictionary"""
28         if var in obs:
29             return {var:1 if val == obs[var] else 0 for val in var.domain}
30         else:
31             if elim_order == None:
32                 elim_order = self.gm.variables
33             projFactors = [self.project_observations(fact, obs)
34                             for fact in self.gm.factors]
35             for v in elim_order:

```

```

36         if v != var and v not in obs:
37             projFactors = self.eliminate_var(projFactors,v)
38         unnorm = factor_times(var,projFactors)
39         p_obs=sum(unnorm)
40         self.display(1,"Unnormalized probs:",unnorm,"Prob obs:",p_obs)
41         return {val:pr/p_obs for val,pr in zip(var.domain, unnorm)}

```

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

probFactors.py — (continued)

```

159 class FactorObserved(Factor):
160     def __init__(self,factor,obs):
161         Factor.__init__(self, [v for v in factor.variables if v not in obs])
162         self.observed = obs
163         self.orig_factor = factor
164
165     def get_value(self,assignment):
166         ass = assignment.copy()
167         for ob in self.observed:
168             ass[ob]=self.observed[ob]
169         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)

```

171 class FactorSum(Factor):
172     def __init__(self,var,factors):
173         self.var_summed_out = var
174         self.factors = factors
175         self.vars = []
176         for fac in factors:
177             for v in fac.variables:
178                 if v is not var and v not in self.vars:
179                     self.vars.append(v)
180         Factor.__init__(self,self.vars)
181         self.values = {}
182
183     def get_value(self,assignment):
184         """lazy implementation: if not saved, compute it. Return saved
            value"""

```

```

185     asst = frozenset(assignment.items())
186     if asst in self.values:
187         return self.values[asst]
188     else:
189         total = 0
190         new_asst = assignment.copy()
191         for val in self.var_summed_out.domain:
192             new_asst[self.var_summed_out] = val
193             total += math.prod(fac.get_value(new_asst) for fac in
194                               self.factors)
195         self.values[asst] = total
196     return total

```

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

```

_____probFactors.py — (continued)_____
197 def factor_times(variable, factors):
198     """when factors are factors just on variable (or on no variables)"""
199     prods = []
200     facs = [f for f in factors if variable in f.variables]
201     for val in variable.domain:
202         ast = {variable:val}
203         prods.append(math.prod(f.get_value(ast) for f in facs))
204     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)_____
43 def project_observations(self, factor, obs):
44     """Returns the resulting factor after observing obs
45
46     obs is a dictionary of {variable:value} pairs.
47     """
48     if any((var in obs) for var in factor.variables):
49         # a variable in factor is observed
50         return FactorObserved(factor, obs)
51     else:
52         return factor
53
54 def eliminate_var(self, factors, var):
55     """Eliminate a variable var from a list of factors.
56     Returns a new set of factors that has var summed out.
57     """
58     self.display(2, "eliminating ", str(var))
59     contains_var = []
60     not_contains_var = []

```

```

61         for fac in factors:
62             if var in fac.variables:
63                 contains_var.append(fac)
64             else:
65                 not_contains_var.append(fac)
66         if contains_var == []:
67             return factors
68         else:
69             newFactor = FactorSum(var,contains_var)
70             self.display(2,"Multiplying:",[str(f) for f in contains_var])
71             self.display(2,"Creating factor:", newFactor)
72             self.display(3, newFactor.to_table()) # factor in detail
73             not_contains_var.append(newFactor)
74         return not_contains_var
75
76 from probGraphicalModels import bn_4ch, A,B,C,D
77 bn_4chv = VE(bn_4ch)
78 ## bn_4chv.query(A,{})
79 ## bn_4chv.query(D,{})
80 ## InferenceMethod.max_display_level = 3 # show more detail in displaying
81 ## InferenceMethod.max_display_level = 1 # show less detail in displaying
82 ## bn_4chv.query(A,{D:True})
83 ## bn_4chv.query(B,{A:True,D:False})
84
85 from probGraphicalModels import
86     bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
87 bn_reportv = VE(bn_report) # answers queries using variable elimination
88 ## bn_reportv.query(Tamper,{})
89 ## InferenceMethod.max_display_level = 0 # show no detail in displaying
90 ## bn_reportv.query(Leaving,{})
91 ## bn_reportv.query(Tamper,{},elim_order=[Smoke,Report,Leaving,Alarm,Fire])
92 ## bn_reportv.query(Tamper,{Report:True})
93 ## bn_reportv.query(Tamper,{Report:True,Smoke:False})
94
95 from probGraphicalModels import bn_sprinkler, Season, Sprinkler, Rained,
96     Grass_wet, Grass_shiny, Shoes_wet
97 bn_sprinklerv = VE(bn_sprinkler)
98 ## bn_sprinklerv.query(Shoes_wet,{})
99 ## bn_sprinklerv.query(Shoes_wet,{Rained:True})
100 ## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:True})
101 ## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:False,Rained:True})
102
103 from probGraphicalModels import bn_lr1, Cough, Fever, Sneeze, Cold, Flu,
104     Covid
105 vdiag = VE(bn_lr1)
106 ## vdiag.query(Cough,{})
107 ## vdiag.query(Cold,{Cough:1,Sneeze:0,Fever:1})
108 ## vdiag.query(Flu,{Cough:0,Sneeze:1,Fever:1})
109 ## vdiag.query(Covid,{Cough:1,Sneeze:0,Fever:1})
110 ## vdiag.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:0})

```

```

108 ## vediag.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:1})
109
110 if __name__ == "__main__":
111     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 _____
11 import random
12 from probGraphicalModels import InferenceMethod
13
14 def sample_one(dist):
15     """returns the index of a single sample from normalized distribution
16     dist."""
17     rand = random.random()*sum(dist.values())
18     cum = 0 # cumulative weights
19     for v in dist:
20         cum += dist[v]
21         if cum > rand:
22             return v

```

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

The method *sample_multiple* generates multiple samples from a distribution defined by *dist*, where *dist* is a $\{value : weight\}$ dictionary, where $weight \geq 0$ and the weights cannot all be zero. This returns a list of values, of length *num_samples*, where each sample is selected with a probability proportional to its weight.

The method generates all of the random numbers, sorts them, and then goes through the distribution once, saving the selected samples.

```

_____probStochSim.py — (continued) _____
23 def sample_multiple(dist, num_samples):
24     """returns a list of num_samples values selected using distribution
25     dist.
26     dist is a {value:weight} dictionary that does not need to be normalized
27     """
28     total = sum(dist.values())
29     rands = sorted(random.random()*total for i in range(num_samples))
30     result = []

```

```

30 dist_items = list(dist.items())
31 cum = dist_items[0][1] # cumulative sum
32 index = 0
33 for r in rand:
34     while r>cum:
35         index += 1
36         cum += dist_items[index][1]
37     result.append(dist_items[index][0])
38 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*:

- n calls to *sample_one*
- sample_multiple*
- Create the cumulative distribution (choose how this is represented) and, for each random number, do a binary search to determine the sample associated with the random number.
- Choose a random number in the range $[i/n, (i+1)/n)$ for each $i \in \text{range}(n)$, where n is the number of samples. Use these as the random numbers to select the particles. (Does this give random samples?)

For each method suggest when it might be the best method.

The *test_sampling* method can be used to generate the statistics from a number of samples. It is useful to see the variability as a function of the number of samples. Try it for few samples and also for many samples.

```

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

```

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).

```

53 class SamplingInferenceMethod(InferenceMethod):
54     """The abstract class of sampling-based belief network inference
        methods"""
55
56     def __init__(self, gm=None):
57         InferenceMethod.__init__(self, gm)
58
59     def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
60         raise NotImplementedError("SamplingInferenceMethod query") #
            abstract

```

9.9.3 Rejection Sampling

```

62 class RejectionSampling(SamplingInferenceMethod):
63     """The class that queries Graphical Models using Rejection Sampling.
64
65     gm is a belief network to query
66     """
67     method_name = "rejection sampling"
68
69     def __init__(self, gm=None):
70         SamplingInferenceMethod.__init__(self, gm)
71
72     def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
73         """computes P(qvar | obs) where
74         qvar is a variable.
75         obs is a {variable:value} dictionary.
76         sample_order is a list of variables where the parents
77         come before the variable.
78         """
79         if sample_order is None:
80             sample_order = self.gm.topological_sort()
81         self.display(2, *sample_order, sep="\t")
82         counts = {val:0 for val in qvar.domain}
83         for i in range(number_samples):
84             rejected = False
85             sample = {}
86             for nvar in sample_order:
87                 fac = self.gm.var2cpt[nvar] #factor with nvar as child
88                 val = sample_one({v:fac.get_value(**sample, nvar=v) for v
89                                 in nvar.domain})
89                 self.display(2, val, end="\t")
90                 if nvar in obs and obs[nvar] != val:
91                     rejected = True
92                     self.display(2, "Rejected")
93                     break
94             sample[nvar] = val

```



```

95         if not rejected:
96             counts[sample[qvar]] += 1
97             self.display(2, "Accepted")
98         tot = sum(counts.values())
99         # As well as the distribution we also include raw counts
100         dist = {c:v/tot if tot>0 else 1/len(qvar.domain) for (c,v) in
               counts.items()}
101         dist["raw_counts"] = counts
102         return dist

```

9.9.4 Likelihood Weighting

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

```

probStochSim.py — (continued)
104 class LikelihoodWeighting(SamplingInferenceMethod):
105     """The class that queries Graphical Models using Likelihood weighting.
106
107     gm is a belief network to query
108     """
109     method_name = "likelihood weighting"
110
111     def __init__(self, gm=None):
112         SamplingInferenceMethod.__init__(self, gm)
113
114     def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
115         """computes P(qvar | obs) where
116         qvar is a variable.
117         obs is a {variable:value} dictionary.
118         sample_order is a list of factors where factors defining the parents
119         come before the factors for the child.
120         """
121         if sample_order is None:
122             sample_order = self.gm.topological_sort()
123         self.display(2, *[v for v in sample_order
124                           if v not in obs], sep="\t")
125         counts = {val:0 for val in qvar.domain}
126         for i in range(number_samples):
127             sample = {}
128             weight = 1.0
129             for nvar in sample_order:
130                 fac = self.gm.var2cpt[nvar]
131                 if nvar in obs:
132                     sample[nvar] = obs[nvar]
133                     weight *= fac.get_value(sample)
134                 else:
135                     val = sample_one({v:fac.get_value(**sample, nvar:v)} for

```

```

136         self.display(2, val, end="\t")
137         sample[nvar] = val
138         counts[sample[qvar]] += weight
139         self.display(2, weight)
140         tot = sum(counts.values())
141         # as well as the distribution we also include the raw counts
142         dist = {c:v/tot for (c,v) in counts.items()}
143         dist["raw_counts"] = counts
144         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)_____
146 class ParticleFiltering(SamplingInferenceMethod):
147     """The class that queries Graphical Models using Particle Filtering.
148
149     gm is a belief network to query
150     """
151     method_name = "particle filtering"
152
153     def __init__(self, gm=None):
154         SamplingInferenceMethod.__init__(self, gm)
155
156     def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
157         """computes P(qvar | obs) where
158         qvar is a variable.
159         obs is a {variable:value} dictionary.
160         sample_order is a list of factors where factors defining the parents
161         come before the factors for the child.
162         """
163         if sample_order is None:
164             sample_order = self.gm.topological_sort()
165         self.display(2, *[v for v in sample_order
166                           if v not in obs], sep="\t")
167         particles = [{ } for i in range(number_samples)]
168         for nvar in sample_order:
169             fac = self.gm.var2cpt[nvar]
170             if nvar in obs:
171                 weights = [fac.get_value(**part, nvar:obs[nvar])] for part
                             in particles]

```

```

172         particles = [{**p, nvar:obs[nvar]} for p in
173                     resample(particles, weights, number_samples)]
174     else:
175         for part in particles:
176             part[nvar] = sample_one({v:fac.get_value(**part,
177                                     nvar:v)} for v in nvar.domain})
178             self.display(2,part[nvar],end="\t")
179 counts = {val:0 for val in qvar.domain}
180 for part in particles:
181     counts[part[qvar]] += 1
182 tot = sum(counts.values())
183 # as well as the distribution we also include the raw counts
184 dist = {c:v/tot for (c,v) in counts.items()}
185 dist["raw_counts"] = counts
186 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)_____
186 def resample(particles, weights, num_samples):
187     """returns num_samples copies of particles resampled according to
188         weights.
189     particles is a list of particles
190     weights is a list of positive numbers, of same length as particles
191     num_samples is n integer
192     """
193     total = sum(weights)
194     rands = sorted(random.random()*total for i in range(num_samples))
195     result = []
196     cum = weights[0] # cumulative sum
197     index = 0
198     for r in rands:
199         while r>cum:
200             index += 1
201             cum += weights[index]
202         result.append(particles[index])
203     return result

```

9.9.6 Examples

```

_____probStochSim.py — (continued)_____
204 from probGraphicalModels import bn_4ch, A,B,C,D
205 bn_4chr = RejectionSampling(bn_4ch)
206 bn_4chL = LikelihoodWeighting(bn_4ch)

```

```

207 ## InferenceMethod.max_display_level = 2 # detailed tracing for all
    inference methods
208 ## bn_4chr.query(A,{})
209 ## bn_4chr.query(C,{})
210 ## bn_4chr.query(A,{C:True})
211 ## bn_4chr.query(B,{A:True,C:False})
212
213 from probGraphicalModels import
    bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
214 bn_reportr = RejectionSampling(bn_report) # answers queries using
    rejection sampling
215 bn_reportL = LikelihoodWeighting(bn_report) # answers queries using
    likelihood weighting
216 bn_reportp = ParticleFiltering(bn_report) # answers queries using particle
    filtering
217 ## bn_reportr.query(Tamper,{})
218 ## bn_reportr.query(Tamper,{})
219 ## bn_reportr.query(Tamper,{Report:True})
220 ## InferenceMethod.max_display_level = 0 # no detailed tracing for all
    inference methods
221 ## bn_reportr.query(Tamper,{Report:True},number_samples=100000)
222 ## bn_reportr.query(Tamper,{Report:True,Smoke:False})
223 ## bn_reportr.query(Tamper,{Report:True,Smoke:False},number_samples=100)
224
225 ## bn_reportL.query(Tamper,{Report:True,Smoke:False},number_samples=100)
226 ## bn_reportL.query(Tamper,{Report:True,Smoke:False},number_samples=100)
227
228 from probGraphicalModels import bn_sprinkler,Season, Sprinkler
229 from probGraphicalModels import Rained, Grass_wet, Grass_shiny, Shoes_wet
230 bn_sprinklerr = RejectionSampling(bn_sprinkler) # answers queries using
    rejection sampling
231 bn_sprinklerL = LikelihoodWeighting(bn_sprinkler) # answers queries using
    rejection sampling
232 bn_sprinklerp = ParticleFiltering(bn_sprinkler) # answers queries using
    particle filtering
233 #bn_sprinklerr.query(Shoes_wet,{Grass_shiny:True,Rained:True})
234 #bn_sprinklerL.query(Shoes_wet,{Grass_shiny:True,Rained:True})
235 #bn_sprinklerp.query(Shoes_wet,{Grass_shiny:True,Rained:True})
236
237 if __name__ == "__main__":
238     InferenceMethod.testIM(RejectionSampling, threshold=0.1)
239     InferenceMethod.testIM(LikelihoodWeighting, threshold=0.1)
240     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) _____
242 #import random
243 #from probGraphicalModels import InferenceMethod
244
245 #from probStochSim import sample_one, SamplingInferenceMethod
246
247 class GibbsSampling(SamplingInferenceMethod):
248     """The class that queries Graphical Models using Gibbs Sampling.
249
250     bn is a graphical model (e.g., a belief network) to query
251     """
252     method_name = "Gibbs sampling"
253
254     def __init__(self, gm=None):
255         SamplingInferenceMethod.__init__(self, gm)
256         self.gm = gm
257
258     def query(self, qvar, obs={}, number_samples=1000, burn_in=100,
259              sample_order=None):
260         """computes P(qvar | obs) where
261         qvar is a variable.
262         obs is a {variable:value} dictionary.
263         sample_order is a list of non-observed variables in order, or
264         if sample_order None, the variables are shuffled at each iteration.
265         """
266         counts = {val:0 for val in qvar.domain}
267         if sample_order is not None:
268             variables = sample_order
269         else:
270             variables = [v for v in self.gm.variables if v not in obs]
271         var_to_factors = {v:set() for v in self.gm.variables}
272         for fac in self.gm.factors:
273             for var in fac.variables:
274                 var_to_factors[var].add(fac)
275         sample = {var:random.choice(var.domain) for var in variables}
276         self.display(2, "Sample:", sample)
277         sample.update(obs)
278         for i in range(burn_in + number_samples):
279             if sample_order == None:
280                 random.shuffle(variables)
281             for var in variables:
282                 # get unnormalized probability distribution of var given its
283                 # neighbours
284                 vardist = {val:1 for val in var.domain}
285                 for val in var.domain:
286                     sample[var] = val

```

```

285         for fac in var_to_factors[var]: # Markov blanket
286             vardist[val] *= fac.get_value(sample)
287             sample[var] = sample_one(vardist)
288         if i >= burn_in:
289             counts[sample[qvar]] +=1
290         tot = sum(counts.values())
291         # as well as the computed distribution, we also include raw counts
292         dist = {c:v/tot for (c,v) in counts.items()}
293         dist["raw_counts"] = counts
294         return dist
295
296 #from probGraphicalModels import bn_4ch, A,B,C,D
297 bn_4chg = GibbsSampling(bn_4ch)
298 ## InferenceMethod.max_display_level = 2 # detailed tracing for all
299     inference methods
300 bn_4chg.query(A,{})
301 ## bn_4chg.query(D,{})
302 ## bn_4chg.query(B,{D:True})
303 ## bn_4chg.query(B,{A:True,C:False})
304
305 from probGraphicalModels import
306     bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
307 bn_reportg = GibbsSampling(bn_report)
308 ## bn_reportg.query(Tamper,{Report:True},number_samples=1000)
309
310 if __name__ == "__main__":
311     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)_____
311 import matplotlib.pyplot as plt
312
313 def plot_stats(method, qvar, qval, obs, number_runs=1000, **queryargs):
314     """Plots a cumulative distribution of the prediction of the model.
315     method is a InferenceMethod (that implements appropriate query(.))
316     plots P(qvar=qval | obs)
317     qvar is the query variable, qval is corresponding value
318     obs is the {variable:value} dictionary representing the observations
319     number_iterations is the number of runs that are plotted
320     **queryargs is the arguments to query (often number_samples for
321         sampling methods)
322     """
323     plt.ion()
324     plt.xlabel("value")
325     plt.ylabel("Cumulative Number")
326     method.max_display_level, prev_mdl = 0, method.max_display_level #no
327         display
328     answers = [method.query(qvar, obs, **queryargs)
329         for i in range(number_runs)]
330     values = [ans[qval] for ans in answers]
331     label = f"{method.method_name} P({qvar}={qval} | {' '.join(f'{var}={val}'
332         for (var, val) in obs.items())))"
333     values.sort()
334     plt.plot(values, range(number_runs), label=label)
335     plt.legend() #loc="upper left")
336     plt.draw()
337     method.max_display_level = prev_mdl # restore display level
338
339 # Try:
340 #
341     plot_stats(bn_reportr, Tamper, True, {Report: True, Smoke: True}, number_samples=1000,
342         number_runs=1000)
343 #
344     plot_stats(bn_reportL, Tamper, True, {Report: True, Smoke: True}, number_samples=1000,
345         number_runs=1000)
346 #

```

```

    plot_stats(bn_reportp, Tamper, True, {Report: True, Smoke: True}, number_samples=1000,
    number_runs=1000)
340 #
    plot_stats(bn_reportr, Tamper, True, {Report: True, Smoke: True}, number_samples=100,
    number_runs=1000)
341 #
    plot_stats(bn_reportL, Tamper, True, {Report: True, Smoke: True}, number_samples=100,
    number_runs=1000)
342 #
    plot_stats(bn_reportg, Tamper, True, {Report: True, Smoke: True}, number_samples=1000,
    number_runs=1000)
343
344 def plot_mult(methods, example, qvar, qval, obs, number_samples=1000,
    number_runs=1000):
345     for method in methods:
346         solver = method(example)
347         if isinstance(method, SamplingInferenceMethod):
348             plot_stats(solver, qvar, qval, obs, number_samples, number_runs)
349         else:
350             plot_stats(solver, qvar, qval, obs, number_runs)
351
352 from probRC import ProbRC
353 # Try following (but it takes a while..)
354 methods =
    [ProbRC, RejectionSampling, LikelihoodWeighting, ParticleFiltering, GibbsSampling]
355 # plot_mult(methods, bn_report, Tamper, True, {Report: True, Smoke: False}, number_samples=100,
    number_runs=1000)
356 #
    plot_mult(methods, bn_report, Tamper, True, {Report: False, Smoke: True}, number_samples=100,
    number_runs=1000)
357
358 # Sprinkler Example:
359 #
    plot_stats(bn_sprinklerr, Shoes_wet, True, {Grass_shiny: True, Rained: True}, number_samples=1000)
360 #
    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.


```

11 import random
12 from probStochSim import sample_one, sample_multiple
13
14 class HMM(object):
15     def __init__(self, states, obsvars, pobs, trans, indist):
16         """A hidden Markov model.
17         states - set of states
18         obsvars - set of observation variables
19         pobs - probability of observations, pobs[i][s] is P(Obs_i=True |
                State=s)
20         trans - transition probability - trans[i][j] gives P(State=j |
                State=i)
21         indist - initial distribution - indist[s] is P(State_0 = s)
22         """
23         self.states = states
24         self.obsvars = obsvars
25         self.pobs = pobs
26         self.trans = trans
27         self.indist = indist

```

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

probHMM.py — (continued)

```

29 # state
30 #     0=middle, 1,2,3 are corners
31 states1 = {'middle', 'c1', 'c2', 'c3'} # states
32 obs1 = {'m1', 'm2', 'm3'} # microphones

```

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

probHMM.py — (continued)

```

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

```

The transition model is as follows: If the animal is in a corner it stays in the same corner with probability 0.80, goes to the middle with probability 0.1

or goes to one of the other corners with probability 0.05 each. If it is in the middle, it stays in the middle with probability 0.7, otherwise it moves to one the corners, each with probability 0.1.

```

probHMM.py — (continued)
41 # trans specifies the dynamics
42 # trans[i] is the distribution over states resulting from state i
43 # trans[i][j] gives P(S=j | S=i)
44 sm=0.7; mmc=0.1          # transition probabilities when in middle
45 sc=0.8; mcm=0.1; mcc=0.05 # transition probabilities when in a corner
46 trans1 = {'middle':{'middle':sm, 'c1':mmc, 'c2':mmc, 'c3':mmc}, # was in
           middle
47           'c1':{'middle':mcm, 'c1':sc, 'c2':mcc, 'c3':mcc}, # was in corner
           1
48           'c2':{'middle':mcm, 'c1':mcc, 'c2':sc, 'c3':mcc}, # was in corner
           2
49           'c3':{'middle':mcm, 'c1':mcc, 'c2':mcc, 'c3':sc}} # was in corner
           3

```

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

```

probHMM.py — (continued)
51 # initially we have a uniform distribution over the animal's state
52 indist1 = {st:1.0/len(states1) for st in states1}
53
54 hmm1 = HMM(states1, obs1, pobs1, trans1, indist1)

```

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)
56 from display import Displayable
57
58 class HMMVEfilter(Displayable):
59     def __init__(self, hmm):
60         self.hmm = hmm
61         self.state_dist = hmm.indist
62
63     def filter(self, obsseq):
64         """updates and returns the state distribution following the
           sequence of
65         observations in obsseq using variable elimination.
66
67         Note that it first advances time.
68         This is what is required if it is called sequentially.
69         If that is not what is wanted initially, do an observe first.
70         """
71         for obs in obsseq:

```

```

72         self.advance()    # advance time
73         self.observe(obs) # observe
74         return self.state_dist
75
76     def observe(self, obs):
77         """updates state conditioned on observations.
78         obs is a list of values for each observation variable"""
79         for i in self.hmm.obsvars:
80             self.state_dist = {st:self.state_dist[st]*(self.hmm.pobs[i][st]
81                                                         if obs[i] else
82                                                         (1-self.hmm.pobs[i][st]))
83                               for st in self.hmm.states}
84         norm = sum(self.state_dist.values()) # normalizing constant
85         self.state_dist = {st:self.state_dist[st]/norm for st in
86                             self.hmm.states}
87         self.display(2,"After observing",obs,"state
88                             distribution:",self.state_dist)
89
90     def advance(self):
91         """advance to the next time"""
92         nextstate = {st:0.0 for st in self.hmm.states} # distribution over
93                     next states
94         for j in self.hmm.states:    # j ranges over next states
95             for i in self.hmm.states: # i ranges over previous states
96                 nextstate[j] += self.hmm.trans[i][j]*self.state_dist[i]
97         self.state_dist = nextstate
98         self.display(2,"After advancing state
99                     distribution:",self.state_dist)

```

The following are some queries for *hmm1*.

```

probHMM.py — (continued)
96 hmm1f1 = HMMVEfilter(hmm1)
97 # hmm1f1.filter([{'m1':0, 'm2':1, 'm3':1}, {'m1':1, 'm2':0, 'm3':1}])
98 ## HMMVEfilter.max_display_level = 2 # show more detail in displaying
99 # hmm1f2 = HMMVEfilter(hmm1)
100 # hmm1f2.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':1, 'm3':0},
101                 {'m1':1, 'm2':0, 'm3':0},
102                 {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
103                 {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
104                 {'m1':0, 'm2':0, 'm3':1}, {'m1':0, 'm2':0, 'm3':1},
105                 {'m1':0, 'm2':0, 'm3':1}])
106 # hmm1f3 = HMMVEfilter(hmm1)
107 # hmm1f3.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
108                 {'m1':1, 'm2':0, 'm3':0}, {'m1':1, 'm2':0, 'm3':1}])
109
110 # How do the following differ in the resulting state distribution?
111 # Note they start the same, but have different initial observations.
112 ## HMMVEfilter.max_display_level = 1 # show less detail in displaying
113 # for i in range(100): hmm1f1.advance()

```

```

111 # hmm1f1.state_dist
112 # for i in range(100): hmm1f3.advance()
113 # hmm1f3.state_dist

```

Exercise 9.6 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_____
11 from probHMM import HMMVEfilter, HMM
12 from display import Displayable
13 import matplotlib.pyplot as plt
14 from matplotlib.widgets import Button, CheckButtons
15
16 class HMM_Controlled(HMM):
17     """A controlled HMM, where the transition probability depends on the
18         action.
19         Instead of the transition probability, it has a function act2trans
20         from action to transition probability.
21         Any algorithms need to select the transition probability according
22         to the action.
23     """
24     def __init__(self, states, obsvars, pobs, act2trans, indist):
25         self.act2trans = act2trans
26         HMM.__init__(self, states, obsvars, pobs, None, indist)
27
28 local_states = list(range(16))
29 door_positions = {2,4,7,11}
30 def prob_door(loc): return 0.8 if loc in door_positions else 0.1
31 local_obs = {'door':[prob_door(i) for i in range(16)]}
32 act2trans = {'right': [[0.1 if next == current
33                        else 0.8 if next == (current+1)%16
34                        else 0.074 if next == (current+2)%16
35                        else 0.002 for next in range(16)] for
36                        current in range(16)],
37               'left': [[0.1 if next == current
38                       else 0.8 if next == (current-1)%16
39                       else 0.074 if next == (current-2)%16
40                       else 0.002 for next in range(16)] for
41                       current in range(16)]}

```

```

39 | 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)_____
40 | class HMM_Local(HMMVEfilter):
41 |     """VE filter for controlled HMMs
42 |     """
43 |     def __init__(self, hmm):
44 |         HMMVEfilter.__init__(self, hmm)
45 |
46 |     def go(self, action):
47 |         self.hmm.trans = self.hmm.act2trans[action]
48 |         self.advance()
49 |
50 | loc_filt = HMM_Local(hmm_16pos)
51 | # loc_filt.observe({'door':True}); loc_filt.go("right");
    |     loc_filt.observe({'door':False}); loc_filt.go("right");
    |     loc_filt.observe({'door':True})
52 | # loc_filt.state_dist

```

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

```

_____probLocalization.py — (continued)_____
54 | class Show_Localization(Displayable):
55 |     def __init__(self, hmm):
56 |         self.hmm = hmm
57 |         self.loc_filt = HMM_Local(hmm)
58 |         fig, (self.ax) = plt.subplots()
59 |         plt.subplots_adjust(bottom=0.2)
60 |         left_but = Button(plt.axes([0.05,0.02,0.1,0.05]), "left")
61 |         left_but.on_clicked(self.left)
62 |         right_but = Button(plt.axes([0.25,0.02,0.1,0.05]), "right")
63 |         right_but.on_clicked(self.right)
64 |         door_but = Button(plt.axes([0.45,0.02,0.1,0.05]), "door")
65 |         door_but.on_clicked(self.door)
66 |         nodoor_but = Button(plt.axes([0.65,0.02,0.1,0.05]), "no door")
67 |         nodoor_but.on_clicked(self.nodoor)
68 |         reset_but = Button(plt.axes([0.85,0.02,0.1,0.05]), "reset")
69 |         reset_but.on_clicked(self.reset)
70 |         #this makes sure y-axis goes to 1, graph overwritten in
    |         draw_dist
71 |         self.draw_dist()
72 |         plt.show()
73 |
74 |     def draw_dist(self):
75 |         self.ax.clear()
76 |         plt.ylim(0,1)

```

```

77     self.ax.set_ylabel("Probability")
78     self.ax.set_xlabel("Location")
79     self.ax.set_title("Location Probability Distribution")
80     self.ax.set_xticks(self.hmm.states)
81     vals = [self.loc_filt.state_dist[i] for i in self.hmm.states]
82     self.bars = self.ax.bar(self.hmm.states, vals, color='black')
83     self.ax.bar_label(self.bars,["{v:.2f}".format(v=v) for v in vals],
84                        padding = 1)
85     plt.draw()
86
87     def left(self,event):
88         self.loc_filt.go("left")
89         self.draw_dist()
90     def right(self,event):
91         self.loc_filt.go("right")
92         self.draw_dist()
93     def door(self,event):
94         self.loc_filt.observe({'door':True})
95         self.draw_dist()
96     def nodoor(self,event):
97         self.loc_filt.observe({'door':False})
98         self.draw_dist()
99     def reset(self,event):
100         self.loc_filt.state_dist = {i:1/16 for i in range(16)}
101         self.draw_dist()
102 # 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) _____
114 from display import Displayable
115 from probStochSim import resample
116
117 class HMMparticleFilter(Displayable):
118     def __init__(self,hmm,number_particles=1000):
119         self.hmm = hmm
120         self.particles = [sample_one(hmm.indist)
121                           for i in range(number_particles)]
122         self.weights = [1 for i in range(number_particles)]
123
124     def filter(self, obsseq):
125         """returns the state distribution following the sequence of
126         observations in obsseq using particle filtering.

```

```

127
128     Note that it first advances time.
129     This is what is required if it is called after previous filtering.
130     If that is not what is wanted initially, do an observe first.
131     """
132     for obs in obsseq:
133         self.advance() # advance time
134         self.observe(obs) # observe
135         self.resample_particles()
136         self.display(2, "After observing", str(obs),
137                     "state distribution:",
138                     self.histogram(self.particles))
139     self.display(1, "Final state distribution:",
140                 self.histogram(self.particles))
141     return self.histogram(self.particles)
142
143 def advance(self):
144     """advance to the next time.
145     This assumes that all of the weights are 1."""
146     self.particles = [sample_one(self.hmm.trans[st])
147                       for st in self.particles]
148
149 def observe(self, obs):
150     """reweighs the particles to incorporate observations obs"""
151     for i in range(len(self.particles)):
152         for obv in obs:
153             if obs[obv]:
154                 self.weights[i] *= self.hmm.pobs[obv][self.particles[i]]
155             else:
156                 self.weights[i] *=
157                     1-self.hmm.pobs[obv][self.particles[i]]
158
159 def histogram(self, particles):
160     """returns list of the probability of each state as represented by
161     the particles"""
162     tot=0
163     hist = {st: 0.0 for st in self.hmm.states}
164     for (st,wt) in zip(self.particles,self.weights):
165         hist[st]+=wt
166         tot += wt
167     return {st:hist[st]/tot for st in hist}
168
169 def resample_particles(self):
170     """resamples to give a new set of particles."""
171     self.particles = resample(self.particles, self.weights,
172                              len(self.particles))
173     self.weights = [1] * len(self.particles)

```

The following are some queries for *hmm1*.

```

171 | hmm1pf1 = HMMparticleFilter(hmm1)
172 | # HMMparticleFilter.max_display_level = 2 # show each step
173 | # hmm1pf1.filter([{'m1':0, 'm2':1, 'm3':1}, {'m1':1, 'm2':0, 'm3':1}])
174 | # hmm1pf2 = HMMparticleFilter(hmm1)
175 | # hmm1pf2.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':1, 'm3':0},
176 | #                 {'m1':1, 'm2':0, 'm3':0},
177 | #                 {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
178 | #                 {'m1':0, 'm2':0, 'm3':0},
179 | #                 {'m1':0, 'm2':0, 'm3':1},
180 | #                 {'m1':0, 'm2':0, 'm3':1},
181 | #                 {'m1':0, 'm2':0, 'm3':1}])
182 | # hmm1pf3 = HMMparticleFilter(hmm1)
183 | # hmm1pf3.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},
184 | #                 {'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)
182 | def simulate(hmm,horizon):
183 |     """returns a pair of (state sequence, observation sequence) of length
184 |         horizon.
185 |         for each time t, the agent is in state_sequence[t] and
186 |         observes observation_sequence[t]
187 |         """
188 |     state = sample_one(hmm.indist)
189 |     obsseq=[]
190 |     stateseq=[]
191 |     for time in range(horizon):
192 |         stateseq.append(state)
193 |         newobs =
194 |             {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
195 |              for obs in hmm.obsvars}
196 |         obsseq.append(newobs)
197 |         state = sample_one(hmm.trans[state])
198 |     return stateseq,obsseq

```



```

199     """returns observation sequence for the state sequence"""
200     obsseq=[]
201     for state in stateseq:
202         newobs =
203             {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
204             for obs in hmm.obsvars}
205         obsseq.append(newobs)
206     return obsseq
207
208 def create_eg(hmm,n):
209     """Create an annotated example for horizon n"""
210     seq,obs = simulate(hmm,n)
211     print("True state sequence:",seq)
212     print("Sequence of observations:\n",obs)
213     hmmfilter = HMMVEfilter(hmm)
214     dist = hmmfilter.filter(obs)
215     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 then move to the next time. This does not allow for arbitrary historical queries (about the past or the future), but can be much simpler. 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.

```

11 from variable import Variable
12 from probGraphicalModels import GraphicalModel, BeliefNetwork
13 from probFactors import Prob, Factor, CPD
14 from probVE import VE
15 from display import Displayable
16
17 class DBNvariable(Variable):
18     """A random variable that incorporates the stage (time)
19
20     A variable can have both a name and an index. The index defaults to 1.
21     """
22     def __init__(self, name, domain=[False, True], index=1):
23         Variable.__init__(self, f"{name}_{index}", domain)
24         self.basename = name
25         self.domain = domain
26         self.index = index
27         self.previous = None
28
29     def __lt__(self, other):
30         if self.name != other.name:
31             return self.name < other.name
32         else:
33             return self.index < other.index
34
35     def __gt__(self, other):
36         return other < self
37
38 def variable_pair(name, domain=[False, True]):
39     """returns a variable and its predecessor. This is used to define
40     2-stage DBNs
41
42     If the name is X, it returns the pair of variables X_prev, X_now"""
43     var_now = DBNvariable(name, domain, index='now')
44     var_prev = DBNvariable(name, domain, index='prev')
45     var_now.previous = var_prev
46     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)
47 class FactorRename(Factor):
48     def __init__(self, fac, renaming):
49         """A renamed factor.
50         fac is a factor
51         renaming is a dictionary of the form {new:old} where old and new
           var variables,
52         where the variables in fac appear exactly once in the renaming
53         """
54         Factor.__init__(self, [n for (n,o) in renaming.items() if o in
           fac.variables])
55         self.orig_fac = fac
56         self.renaming = renaming
57
58     def get_value(self, assignment):
59         return self.orig_fac.get_value({self.renaming[var]:val
60                                         for (var,val) in assignment.items()
61                                         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)
63 class CPDrename(FactorRename, CPD):
64     def __init__(self, cpd, renaming):
65         renaming_inverse = {old:new for (new,old) in renaming.items()}
66         CPD.__init__(self, renaming_inverse[cpd.child], [renaming_inverse[p]
           for p in cpd.parents])
67         self.orig_fac = cpd
68         self.renaming = renaming

```

```

probDBN.py — (continued)
70 class DBN(Displayable):
71     """The class of stationary Dynamic Belief networks.
72     * name is the DBN name
73     * vars_now is a list of current variables (each must have
74     previous variable).
75     * transition_factors is a list of factors for P(X|parents) where X
76     is a current variable and parents is a list of current or previous
       variables.
77     * init_factors is a list of factors for P(X|parents) where X is a
78     current variable and parents can only include current variables
79     The graph of transition factors + init factors must be acyclic.
80
81     """
82     def __init__(self, title, vars_now, transition_factors=None,
           init_factors=None):
83         self.title = title
84         self.vars_now = vars_now

```

```

85     self.vars_prev = [v.previous for v in vars_now]
86     self.transition_factors = transition_factors
87     self.init_factors = init_factors
88     self.var_index = {} # var_index[v] is the index of variable v
89     for i,v in enumerate(vars_now):
90         self.var_index[v]=i

```

Here is a 3 variable DBN:

```

probDBN.py — (continued)
92 A0,A1 = variable_pair("A", domain=[False,True])
93 B0,B1 = variable_pair("B", domain=[False,True])
94 C0,C1 = variable_pair("C", domain=[False,True])
95
96 # dynamics
97 pc = Prob(C1,[B1,C0],[[0.03,0.97],[0.38,0.62]],[[0.23,0.77],[0.78,0.22]])
98 pb = Prob(B1,[A0,A1],[[0.5,0.5],[0.77,0.23]],[[0.4,0.6],[0.83,0.17]])
99 pa = Prob(A1,[A0,B0],[[0.1,0.9],[0.65,0.35]],[[0.3,0.7],[0.8,0.2]])
100
101 # initial distribution
102 pa0 = Prob(A1,[],[0.9,0.1])
103 pb0 = Prob(B1,[A1],[0.3,0.7],[0.8,0.2])
104 pc0 = Prob(C1,[],[0.2,0.8])
105
106 dbn1 = DBN("Simple DBN",[A1,B1,C1],[pa,pb,pc],[pa0,pb0,pc0])

```

Here is the animal example

```

probDBN.py — (continued)
108 from probHMM import closeMic, farMic, midMic, sm, mmc, sc, mcm, mcc
109
110 Pos_0,Pos_1 = variable_pair("Position",domain=[0,1,2,3])
111 Mic1_0,Mic1_1 = variable_pair("Mic1")
112 Mic2_0,Mic2_1 = variable_pair("Mic2")
113 Mic3_0,Mic3_1 = variable_pair("Mic3")
114
115 # conditional probabilities - see hmm for the values of sm,mmc, etc
116 ppos = Prob(Pos_1, [Pos_0],
117             [[sm, mmc, mmc, mmc], #was in middle
118              [mcm, sc, mcc, mcc], #was in corner 1
119              [mcm, mcc, sc, mcc], #was in corner 2
120              [mcm, mcc, mcc, sc]]) #was in corner 3
121 pm1 = Prob(Mic1_1, [Pos_1], [[1-midMic, midMic], [1-closeMic, closeMic],
122                             [1-farMic, farMic], [1-farMic, farMic]])
123 pm2 = Prob(Mic2_1, [Pos_1], [[1-midMic, midMic], [1-farMic, farMic],
124                             [1-closeMic, closeMic], [1-farMic, farMic]])
125 pm3 = Prob(Mic3_1, [Pos_1], [[1-midMic, midMic], [1-farMic, farMic],
126                             [1-farMic, farMic], [1-closeMic, closeMic]])
127 ipos = Prob(Pos_1,[], [0.25, 0.25, 0.25, 0.25])
128 dbn_an =DBN("Animal DBN",[Pos_1,Mic1_1,Mic2_1,Mic3_1],
129             [ppos, pm1, pm2, pm3],
130             [ipos, pm1, pm2, pm3])

```

9.11.2 Unrolling DBNs

```

132 class BNfromDBN(BeliefNetwork):
133     """Belief Network unrolled from a dynamic belief network
134     """
135
136     def __init__(self,dbn,horizon):
137         """dbn is the dynamic belief network being unrolled
138         horizon>0 is the number of steps (so there will be horizon+1
139         variables for each DBN variable.
140         """
141         self.name2var = {var.basename:
142             [DBNvariable(var.basename,var.domain,index) for index in
143              range(horizon+1)]
144             for var in dbn.vars_now}
145         self.display(1,f"name2var={self.name2var}")
146         variables = {v for vs in self.name2var.values() for v in vs}
147         self.display(1,f"variables={variables}")
148         bnfactors = {CPDrename(fac,{self.name2var[var.basename][0]:var
149             for var in fac.variables})
150             for fac in dbn.init_factors}
151         bnfactors |= {CPDrename(fac,{self.name2var[var.basename][i]:var
152             for var in fac.variables if
153                 var.index=='prev'}
154             | {self.name2var[var.basename][i+1]:var
155                 for var in fac.variables if
156                     var.index=='now'}}
157         for fac in dbn.transition_factors
158             for i in range(horizon)}
159         self.display(1,f"bnfactors={bnfactors}")
160         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).

```

157 # Try
158 #from probRC import ProbRC
159 #bn = BNfromDBN(dbn1,2) # construct belief network
160 #drc = ProbRC(bn) # initialize recursive conditioning
161 #B2 = bn.name2var['B'][2]
162 #drc.query(B2) #P(B2)
163 #drc.query(bn.name2var['B'][1],{bn.name2var['B'][0]:True,bn.name2var['C'][1]:False})
164 #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.

```

164 class DBNVEfilter(VE):
165     def __init__(self,dbn):
166         self.dbn = dbn
167         self.current_factors = dbn.init_factors
168         self.current_obs = {}
169
170     def observe(self, obs):
171         """updates the current observations with obs.
172         obs is a variable:value dictionary where variable is a current
173         variable.
174         """
175         assert all(self.current_obs[var]==obs[var] for var in obs
176                   if var in self.current_obs),"inconsistent current
177                   observations"
178         self.current_obs.update(obs) # note 'update' is a dict method
179
180     def query(self,var):
181         """returns the posterior probability of current variable var"""
182         return
183             VE(GraphicalModel(self.dbn.title,self.dbn.vars_now,self.current_factors)).query(var,self.c
184
185     def advance(self):
186         """advance to the next time"""
187         prev_factors = [self.make_previous(fac) for fac in
188             self.current_factors]
189         prev_obs = {var.previous:val for var,val in
190             self.current_obs.items()}
191         two_stage_factors = prev_factors + self.dbn.transition_factors
192         self.current_factors =
193             self.elim_vars(two_stage_factors,self.dbn.vars_prev,prev_obs)
194         self.current_obs = {}
195
196     def make_previous(self,fac):
197         """Creates new factor from fac where the current variables in fac
198         are renamed to previous variables.
199         """
200         return FactorRename(fac, {var.previous:var for var in
201             fac.variables})
202
203     def elim_vars(self,factors, vars, obs):
204         for var in vars:
205             if var in obs:
206                 factors = [self.project_observations(fac,obs) for fac in
207                     factors]
208             else:
209                 factors = self.eliminate_var(factors, var)
210         return factors

```

Example queries:

```
_____probDBN.py — (continued) _____  
205 #df = DBNVEfilter(dbn1)  
206 #df.observe({B1:True}); df.advance(); df.observe({C1:False})  
207 #df.query(B1) #P(B1|B0,C1)  
208 #df.advance(); df.query(B1)  
209 #dfa = DBNVEfilter(dbn_an)  
210 # dfa.observe({Mic1_1:0, Mic2_1:1, Mic3_1:1})  
211 # dfa.advance()  
212 # dfa.observe({Mic1_1:1, Mic2_1:0, Mic3_1:1})  
213 # dfa.query(Pos_1)
```


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 *i*th feature in class *i* is

$$\frac{\text{feature_sum}[i][c]}{\text{class_counts}[c]}$$

The class is initialized by randomly assigning examples to classes, and updating the statistics for *class_counts* and *feature_sum*.

```
learnKMeans.py — k-means learning
11 from learnProblem import Data_set, Learner, Data_from_file
12 import random
13 import matplotlib.pyplot as plt
14
15 class K_means_learner(Learner):
16     def __init__(self, dataset, num_classes):
17         self.dataset = dataset
18         self.num_classes = num_classes
19         self.random_initialize()
20
21     def random_initialize(self):
```

```

22     # class_counts[c] is the number of examples with class=c
23     self.class_counts = [0]*self.num_classes
24     # feature_sum[i][c] is the sum of the values of feature i for class
      c
25     self.feature_sum = [[0]*self.num_classes
26                        for feat in self.dataset.input_features]
27     for eg in self.dataset.train:
28         cl = random.randrange(self.num_classes) # assign eg to random
           class
29         self.class_counts[cl] += 1
30         for (ind,feat) in enumerate(self.dataset.input_features):
31             self.feature_sum[ind][cl] += feat(eg)
32     self.num_iterations = 0
33     self.display(1,"Initial class counts: ",self.class_counts)

```

The distance from (the mean of) a class to an example is the sum, over all features, of the sum-of-squares differences of the class mean and the example value.

```

_____learnKMeans.py — (continued) _____
35     def distance(self,cl,eg):
36         """distance of the eg from the mean of the class"""
37         return sum( (self.class_prediction(ind,cl)-feat(eg))**2
38                    for (ind,feat) in
                        enumerate(self.dataset.input_features))
39
40     def class_prediction(self,feat_ind,cl):
41         """prediction of the class cl on the feature with index feat_ind"""
42         if self.class_counts[cl] == 0:
43             return 0 # there are no examples so we can choose any value
44         else:
45             return self.feature_sum[feat_ind][cl]/self.class_counts[cl]
46
47     def class_of_eg(self,eg):
48         """class to which eg is assigned"""
49         return (min((self.distance(cl,eg),cl)
50                    for cl in range(self.num_classes)))[1]
51         # second element of tuple, which is a class with minimum
           distance

```

One step of k-means updates the *class_counts* and *feature_sum*. It uses the old values to determine the classes, and so the new values for *class_counts* and *feature_sum*. At the end it determines whether the values of these have changes, and then replaces the old ones with the new ones. It returns an indicator of whether the values are stable (have not changed).

```

_____learnKMeans.py — (continued) _____
53     def k_means_step(self):
54         """Updates the model with one step of k-means.
55         Returns whether the assignment is stable.
56         """

```

```

57         new_class_counts = [0]*self.num_classes
58         # feature_sum[i][c] is the sum of the values of feature i for class
           c
59         new_feature_sum = [[0]*self.num_classes
60                             for feat in self.dataset.input_features]
61         for eg in self.dataset.train:
62             cl = self.class_of_eg(eg)
63             new_class_counts[cl] += 1
64             for (ind,feat) in enumerate(self.dataset.input_features):
65                 new_feature_sum[ind][cl] += feat(eg)
66         stable = (new_class_counts == self.class_counts) and
           (self.feature_sum == new_feature_sum)
67         self.class_counts = new_class_counts
68         self.feature_sum = new_feature_sum
69         self.num_iterations += 1
70         return stable
71
72
73     def learn(self,n=100):
74         """do n steps of k-means, or until convergence"""
75         i=0
76         stable = False
77         while i<n and not stable:
78             stable = self.k_means_step()
79             i += 1
80             self.display(1,"Iteration",self.num_iterations,
81                         "class counts: ",self.class_counts,"
                        Stable=",stable)
82
83         return stable
84
85     def show_classes(self):
86         """sorts the data by the class and prints in order.
87         For visualizing small data sets
88         """
89         class_examples = [[] for i in range(self.num_classes)]
90         for eg in self.dataset.train:
91             class_examples[self.class_of_eg(eg)].append(eg)
92         print("Class","Example",sep='\t')
93         for cl in range(self.num_classes):
94             for eg in class_examples[cl]:
95                 print(cl,*eg,sep='\t')
96
97     def plot_error(self, maxstep=20):
98         """Plots the sum-of-squares error as a function of the number of
99         steps"""
100         plt.ion()
101         plt.xlabel("step")
102         plt.ylabel("Ave sum-of-squares error")
103         train_errors = []
104         if self.dataset.test:

```

```

103         test_errors = []
104         for i in range(maxstep):
105             self.learn(1)
106             train_errors.append( sum(self.distance(self.class_of_eg(eg),eg)
107                                   for eg in self.dataset.train)
108                               /len(self.dataset.train))
109             if self.dataset.test:
110                 test_errors.append(
111                     sum(self.distance(self.class_of_eg(eg),eg)
112                           for eg in self.dataset.test)
113                     /len(self.dataset.test))
114             plt.plot(range(1,maxstep+1),train_errors,
115                     label=str(self.num_classes)+" classes. Training set")
116             if self.dataset.test:
117                 plt.plot(range(1,maxstep+1),test_errors,
118                         label=str(self.num_classes)+" classes. Test set")
119             plt.legend()
120             plt.draw()
121 %data = Data_from_file('data/emdata1.csv', num_train=10,
122                       target_index=2000) % trivial example
123 data = Data_from_file('data/emdata2.csv', num_train=10, target_index=2000)
124 %data = Data_from_file('data/emdata0.csv', num_train=14,
125                       target_index=2000) % example from textbook
126 kml = K_means_learner(data,2)
127 num_iter=4
128 print("Class assignment after",num_iter,"iterations:")
129 kml.learn(num_iter); kml.show_classes()
130
131 # Plot the error
132 # km2=K_means_learner(data,2); km2.plot_error(20) # 2 classes
133 # km3=K_means_learner(data,3); km3.plot_error(20) # 3 classes
134 # km13=K_means_learner(data,13); km13.plot_error(20) # 13 classes
135
136 # data = Data_from_file('data/carbool.csv',
137                       target_index=2000,boolean_features=True)
138 # kml = K_means_learner(data,3)
139 # kml.learn(20); kml.show_classes()
140 # km3=K_means_learner(data,3); km3.plot_error(20) # 3 classes
141 # 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 $[0, \text{num_classes})$. i is an index of a feature, so $\text{feat}[i]$ is the i th feature, and a feature is a function from tuples to values. val is a value of a feature.

A model consists of 2 lists, which form the sufficient statistics:

- *class_counts* is a list such that $\text{class_counts}[c]$ is the number of tuples with $\text{class} = c$, where each tuple is weighted by its probability, i.e.,

$$\text{class_counts}[c] = \sum_{t:\text{class}(t)=c} P(t)$$

- *feature_counts* is a list such that $\text{feature_counts}[i][\text{val}][c]$ is the weighted count of the number of tuples t with $\text{feat}[i](t) = \text{val}$ and $\text{class}(t) = c$, each tuple is weighted by its probability, i.e.,

$$\text{feature_counts}[i][\text{val}][c] = \sum_{t:\text{feat}[i](t)=\text{val} \text{ and } \text{class}(t)=c} P(t)$$

```

learnEM.py — EM Learning
11 from learnProblem import Data_set, Learner, Data_from_file
12 import random
13 import math
14 import matplotlib.pyplot as plt
15
16 class EM_learner(Learner):
17     def __init__(self, dataset, num_classes):
18         self.dataset = dataset
19         self.num_classes = num_classes
20         self.class_counts = None
21         self.feature_counts = None

```

The function *em_step* goes 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)
23 def em_step(self, orig_class_counts, orig_feature_counts):
24     """updates the model."""
25     class_counts = [0]*self.num_classes
26     feature_counts = [{val:[0]*self.num_classes
27                       for val in feat.frange}
28                      for feat in self.dataset.input_features]
29     for tple in self.dataset.train:
30         if orig_class_counts: # a model exists
31             tple_class_dist = self.prob(tple, orig_class_counts,
32                                         orig_feature_counts)
33         else: # initially, with no model, return a random
34             distribution
35             tple_class_dist = random_dist(self.num_classes)
36         for cl in range(self.num_classes):
37             class_counts[cl] += tple_class_dist[cl]
38             for (ind, feat) in enumerate(self.dataset.input_features):
39                 feature_counts[ind][feat(tple)][cl] += tple_class_dist[cl]
40     return class_counts, feature_counts

```

prob computes the probability of a class *c* for a tuple *tple*, given the current statistics.

$$\begin{aligned}
 P(c \mid tple) &\propto P(c) * \prod_i P(X_i = tple(i) \mid c) \\
 &= \frac{class_counts[c]}{len(self.dataset)} * \prod_i \frac{feature_counts[i][feat_i(tple)][c]}{class_counts[c]} \\
 &\propto \frac{\prod_i feature_counts[i][feat_i(tple)][c]}{class_counts[c]^{|feats|-1}}
 \end{aligned}$$

The last step is because $len(self.dataset)$ is a constant (independent of *c*). $class_counts[c]$ can be taken out of the product, but needs to be raised to the power of the number of features, and one of them cancels.

```

learnEM.py — (continued)
40 def prob(self, tple, class_counts, feature_counts):
41     """returns a distribution over the classes for tuple tple in the
42     model defined by the counts
43     """
44     feats = self.dataset.input_features
45     unnorm = [prod(feature_counts[i][feat(tple)][c]
46                  for (i, feat) in enumerate(feats))
47              /(class_counts[c]**(len(feats)-1))
48              for c in range(self.num_classes)]
49     thesum = sum(unnorm)
50     return [un/thesum for un in unnorm]

```

learn does *n* steps of EM:

```

learnEM.py — (continued)

```

```

51 | def learn(self,n):
52 |     """do n steps of em"""
53 |     for i in range(n):
54 |         self.class_counts,self.feature_counts =
55 |             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)
57 | def show_class(self,c):
58 |     """sorts the data by the class and prints in order.
59 |     For visualizing small data sets
60 |     """
61 |     sorted_data =
62 |         sorted((self.prob(tpl,self.class_counts,self.feature_counts)[c],
63 |                ind, # preserve ordering for equal
64 |                probabilities
65 |                tpl)
66 |               for (ind,tpl) in enumerate(self.dataset.train))
67 |     for cc,r,tpl in sorted_data:
68 |         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:

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

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

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

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

```

learnEM.py — (continued)
68 | def logloss(self,tple):
69 |     """returns the logloss of the prediction on tple, which is
70 |     -log(P(tple))
71 |     based on the current class counts and feature counts
72 |     """
73 |     feats = self.dataset.input_features
74 |     res = 0
75 |     cc = self.class_counts
76 |     fc = self.feature_counts

```

```

76     for c in range(self.num_classes):
77         res += prod(fc[i][feat(tple)][c]
78                     for (i, feat) in
79                         enumerate(feats))/(cc[c]**(len(feats)-1))
79     if res>0:
80         return -math.log2(res/len(self.dataset.train))
81     else:
82         return float("inf") #infinity
83
84 def plot_error(self, maxstep=20):
85     """Plots the logloss error as a function of the number of steps"""
86     plt.ion()
87     plt.xlabel("step")
88     plt.ylabel("Ave Logloss (bits)")
89     train_errors = []
90     if self.dataset.test:
91         test_errors = []
92     for i in range(maxstep):
93         self.learn(1)
94         train_errors.append( sum(self.logloss(tple) for tple in
95                                self.dataset.train)
96                             /len(self.dataset.train))
97         if self.dataset.test:
98             test_errors.append( sum(self.logloss(tple) for tple in
99                                   self.dataset.test)
100                                /len(self.dataset.test))
101     plt.plot(range(1,maxstep+1),train_errors,
102             label=str(self.num_classes)+" classes. Training set")
103     if self.dataset.test:
104         plt.plot(range(1,maxstep+1),test_errors,
105                 label=str(self.num_classes)+" classes. Test set")
106     plt.legend()
107     plt.draw()
108
109 def prod(L):
110     """returns the product of the elements of L"""
111     res = 1
112     for e in L:
113         res *= e
114     return res
115
116 def random_dist(k):
117     """generate k random numbers that sum to 1"""
118     res = [random.random() for i in range(k)]
119     s = sum(res)
120     return [v/s for v in res]
121
122 data = Data_from_file('data/emdata2.csv', num_train=10, target_index=2000)
123 eml = EM_learner(data,2)
124 num_iter=2

```



```

123 print("Class assignment after",num_iter,"iterations:")
124 eml.learn(num_iter); eml.show_class(0)
125
126 # Plot the error
127 # em2=EM_learner(data,2); em2.plot_error(40) # 2 classes
128 # em3=EM_learner(data,3); em3.plot_error(40) # 3 classes
129 # em13=EM_learner(data,13); em13.plot_error(40) # 13 classes
130
131 # data = Data_from_file('data/carbool.csv',
132 #                       target_index=2000,boolean_features=False)
133 # [f.frange for f in data.input_features]
134 # eml = EM_learner(data,3)
135 # eml.learn(20); eml.show_class(0)
136 # em3=EM_learner(data,3); em3.plot_error(60) # 3 classes
137 # 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.)

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_____
11 from probGraphicalModels import InferenceMethod, BeliefNetwork
12 from probFactors import CPD, ConstantCPD
13
14 def queryDo(self, qvar, obs={}, do={}):
15     assert isinstance(self.gm, BeliefNetwork), "Do only applies to belief
        networks"
16     if do=={}:
17         return self.query(qvar, obs)
18     else:
19         newfacs = ({f for (ch,f) in self.gm.var2cpt.items() if ch not in
            do} |
20                     {ConstantCPD(v,c) for (v,c) in do.items()})
21         self.modBN = BeliefNetwork(self.gm.title+"(mod)",
            self.gm.variables, newfacs)
22         oldBN, self.gm = self.gm, self.modBN
23         result = self.query(qvar, obs)
24         self.gm = oldBN # restore original
25         return result
26
27 InferenceMethod.queryDo = queryDo
```

```
_____probDo.py — (continued)_____
29 from probRC import ProbRC
```

```

30
31 from probGraphicalModels import bn_sprinkler, Season, Sprinkler, Rained,
    Grass_wet, Grass_shiny, Shoes_wet, bn_sprinkler_soff
32 bn_sprinklerv = ProbRC(bn_sprinkler)
33 ## bn_sprinklerv.queryDo(Shoes_wet)
34 ## bn_sprinklerv.queryDo(Shoes_wet, obs={Sprinkler:"off"})
35 ## bn_sprinklerv.queryDo(Shoes_wet, do={Sprinkler:"off"})
36 ## ProbRC(bn_sprinkler_soff).query(Shoes_wet) # should be same as previous
    case
37 ## bn_sprinklerv.queryDo(Season, obs={Sprinkler:"off"})
38 ## bn_sprinklerv.queryDo(Season, do={Sprinkler:"off"})

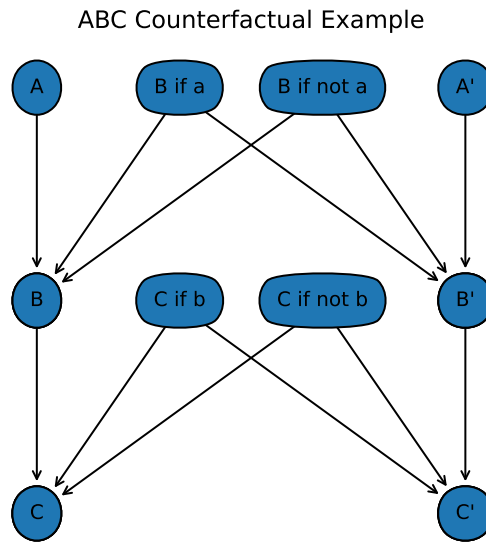
```

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) _____
40 from probVariables import Variable
41 from probFactors import Prob
42 from probGraphicalModels import boolean, BeliefNetwork
43
44 Takes_Marijuana = Variable("Takes_Marijuana", boolean, position=(0.1,0.5))
45 Drug_Prone = Variable("Drug_Prone", boolean, position=(0.1,0.5)) #
    (0.5,0.9))
46 Side_Effects = Variable("Side_Effects", boolean, position=(0.1,0.5)) #
    (0.5,0.1))
47 Takes_Hard_Drugs = Variable("Takes_Hard_Drugs", boolean,
    position=(0.9,0.5))
48
49 p_tm = Prob(Takes_Marijuana, [Drug_Prone], [[0.98, 0.02], [0.2, 0.8]])
50 p_dp = Prob(Drug_Prone, [], [0.8, 0.2])
51 p_be = Prob(Side_Effects, [Takes_Marijuana], [[1, 0], [0.4, 0.6]])
52 p_thd = Prob(Takes_Hard_Drugs, [Side_Effects, Drug_Prone],
53             # Drug_Prone=False Drug_Prone=True
54             [[0.999, 0.001], [0.6, 0.4]], # Side_Effects=False
55             [[0.99999, 0.00001], [0.995, 0.005]]]) # Side_Effects=True
56
57 drugs = BeliefNetwork("Gateway Drug?",
58                       [Takes_Marijuana, Drug_Prone, Side_Effects, Takes_Hard_Drugs],
59                       [p_tm, p_dp, p_be, p_thd])
60
61 drugsq = ProbRC(drugs)
62 # drugsq.queryDo(Takes_Hard_Drugs)
63 # drugsq.queryDo(Takes_Hard_Drugs, obs = {Takes_Marijuana: True})
64 # drugsq.queryDo(Takes_Hard_Drugs, obs = {Takes_Marijuana: False})
65 # drugsq.queryDo(Takes_Hard_Drugs, do = {Takes_Marijuana: True})
66 # 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 $A=\text{true}, C=\text{true}$ is observed; what is the probability of C is A were false. See Figure 11.1.

```

11 from probVariables import Variable
12 from probFactors import Prob
13 from probGraphicalModels import boolean, BeliefNetwork
14 from probRC import ProbRC
15 from probDo import queryDo
16
17 # without a deterministic system
18 Ap = Variable("Ap", boolean, position=(0.2,0.8))
19 Bp = Variable("Bp", boolean, position=(0.2,0.4))
20 Cp = Variable("Cp", boolean, position=(0.2,0.0))
21 p_Ap = Prob(Ap, [], [0.5,0.5])
22 p_Bp = Prob(Bp, [Ap], [[0.6,0.4], [0.6,0.4]]) # does not depend on A!
23 p_Cp = Prob(Cp, [Bp], [[0.2,0.8], [0.9,0.1]])
24 abcSimple = BeliefNetwork("ABC Simple",
25                           [Ap,Bp,Cp],
26                           [p_Ap, p_Bp, p_Cp])
27 ABCsimpq = ProbRC(abcSimple)
28 # ABCsimpq.show_post(obs = {Ap:True, Cp:True})

```

```

29 |
30 | # as a deterministic system with independent noise
31 | A = Variable("A", boolean, position=(0.2,0.8))
32 | B = Variable("B", boolean, position=(0.2,0.4))
33 | C = Variable("C", boolean, position=(0.2,0.0))
34 | Aprime = Variable("A'", boolean, position=(0.8,0.8))
35 | Bprime = Variable("B'", boolean, position=(0.8,0.4))
36 | Cprime = Variable("C'", boolean, position=(0.8,0.0))
37 | BifA = Variable("B if a", boolean, position=(0.4,0.8))
38 | BifnA = Variable("B if not a", boolean, position=(0.6,0.8))
39 | CifB = Variable("C if b", boolean, position=(0.4,0.4))
40 | CifnB = Variable("C if not b", boolean, position=(0.6,0.4))
41 |
42 | p_A = Prob(A, [], [0.5,0.5])
43 | p_B = Prob(B, [A, BifA, BifnA], [[[[1,0],[0,1]],[[1,0],[0,1]]], # A=0
44 |                                     [[1,0],[1,0]],[[0,1],[0,1]]]) # A=1
45 | p_C = Prob(C, [B, CifB, CifnB], [[[[1,0],[0,1]],[[1,0],[0,1]]], # B=0
46 |                                     [[1,0],[1,0]],[[0,1],[0,1]]]) # B=1
47 | p_Aprime = Prob(Aprime,[], [0.6,0.4])
48 | p_Bprime = Prob(Bprime, [Aprime, BifA, BifnA],
49 |                                     [[[[1,0],[0,1]],[[1,0],[0,1]]], # A=0
50 |                                     [[1,0],[1,0]],[[0,1],[0,1]]]) # A=1
51 | p_Cprime = Prob(Cprime, [Bprime, CifB, CifnB],
52 |                                     [[[[1,0],[0,1]],[[1,0],[0,1]]], # B=0
53 |                                     [[1,0],[1,0]],[[0,1],[0,1]]]) # B=1
54 | p_bifa = Prob(BifA, [], [0.6,0.4]) # Does not actually depend on A!!!
55 | p_bifna = Prob(BifnA, [], [0.6,0.4])
56 | p_cifb = Prob(CifB, [], [0.9,0.1])
57 | p_cifnb = Prob(CifnB, [], [0.2,0.8])
58 |
59 | abcCounter = BeliefNetwork("ABC Counterfactual Example",
60 |                             [A,B,C,Aprime,Bprime,Cprime,BifA, BifnA, CifB,
61 |                               CifnB],
62 |                             [p_A,p_B,p_C,p_Aprime,p_Bprime, p_Cprime, p_bifa,
63 |                               p_bifna, p_cifb, p_cifnb])
64 |
65 | abcq = ProbRC(abcCounter)
66 | # abcq.queryDo(Cprime, obs = {Aprime:False, A:True})
67 | # abcq.queryDo(Cprime, obs = {C:True, Aprime:False})
68 | # abcq.queryDo(Cprime, obs = {A:True, C:True, Aprime:False})
69 | # abcq.queryDo(Cprime, obs = {A:True, C:True, Aprime:False})
70 | # abcq.queryDo(Cprime, obs = {A:False, C:True, Aprime:False})
71 | # abcq.queryDo(CifB, obs = {C:True, Aprime:False})
72 | # abcq.queryDo(CifnB, obs = {C:True, Aprime:False})
73 |
74 | # abcq.show_post(obs = {})
75 | # abcq.show_post(obs = {Aprime:False, A:True})
76 | # abcq.show_post(obs = {A:True, C:True, Aprime:False})
77 | # abcq.show_post(obs = {A:True, C:True, Aprime:True})

```

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

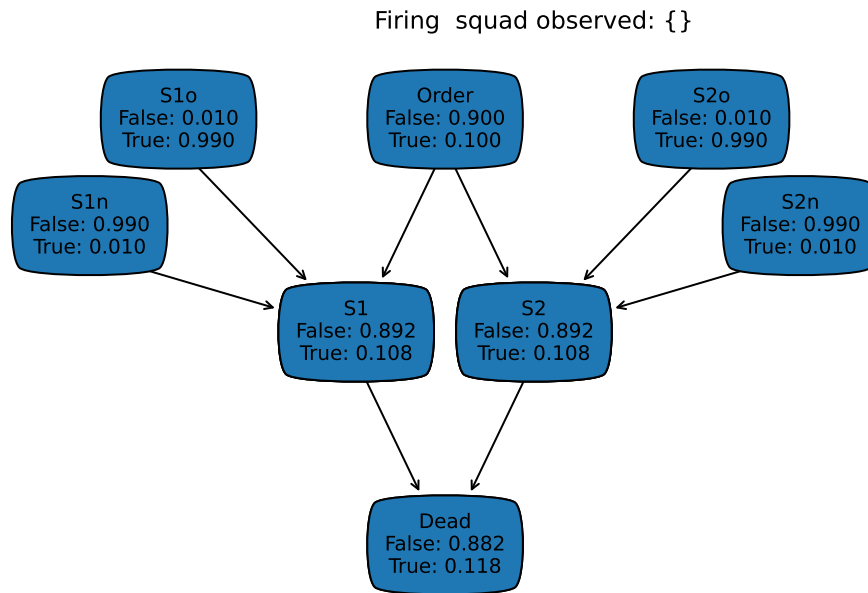


Figure 11.2: Firing squad belief network

```

probCounterfactual.py — (continued)
75 Order = Variable("Order", boolean, position=(0.4,0.8))
76 S1 = Variable("S1", boolean, position=(0.3,0.4))
77 S1o = Variable("S1o", boolean, position=(0.1,0.8))
78 S1n = Variable("S1n", boolean, position=(0.0,0.6))
79 S2 = Variable("S2", boolean, position=(0.5,0.4))
80 S2o = Variable("S2o", boolean, position=(0.7,0.8))
81 S2n = Variable("S2n", boolean, position=(0.8,0.6))
82 Dead = Variable("Dead", boolean, position=(0.4,0.0))
83
84 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
85
86 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
87
88 p_dead = Prob(Dead, [S1,S2], [[[1,0],[0,1]],[[0,1],[0,1]]])
89 p_order = Prob(Order, [], [0.9, 0.1])
90 p_s1o = Prob(S1o, [], [0.01, 0.99])
91 p_s1n = Prob(S1n, [], [0.99, 0.01])
92 p_s2o = Prob(S2o, [], [0.01, 0.99])
93 p_s2n = Prob(S2n, [], [0.99, 0.01])
94
95 firing_squad = BeliefNetwork("Firing squad",

```

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


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-----
11 from probGraphicalModels import GraphicalModel, BeliefNetwork
12 from probFactors import Factor, CPD, TabFactor, factor_times, Prob
13 from probVariables import Variable
14 import matplotlib.pyplot as plt
15
16 class Utility(Factor):
17     """A factor defining a utility"""
18     pass
19
20 class UtilityTable(TabFactor, Utility):
21     """A factor defining a utility using a table"""
22     def __init__(self, vars, table, position=None):
23         """Creates a factor on vars from the table.
24         The table is ordered according to vars.
25         """
26         TabFactor.__init__(self, vars, table)
27         self.position = position
```

A **decision variable** is like a random variable with a string name, and a domain, which is a list of possible values. The decision variable also includes the parents, a list of the variables whose value will be known when the decision is made. It also includes a position, which is only used for plotting.

```

decnNetworks.py — (continued)
29 class DecisionVariable(Variable):
30     def __init__(self, name, domain, parents, position=None):
31         Variable.__init__(self, name, domain, position)
32         self.parents = parents
33         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)
35 class DecisionNetwork(BeliefNetwork):
36     def __init__(self, title, vars, factors):
37         """vars is a list of variables
38         factors is a list of factors (instances of CPD and Utility)
39         """
40         GraphicalModel.__init__(self, title, vars, factors) # don't call
41         init for BeliefNetwork
42         self.var2parents = ({v : v.parents for v in vars if
43                             isinstance(v,DecisionVariable)}
44                             | {f.child:f.parents for f in factors if
45                               isinstance(f,CPD)})
46         self.children = {n:[] for n in self.variables}
47         for v in self.var2parents:
48             for par in self.var2parents[v]:
49                 self.children[par].append(v)
50         self.utility_factor = [f for f in factors if
51                               isinstance(f,Utility)][0]
52         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)
50 def split_order(self):
51     so = []
52     tops = self.topological_sort()
53     for v in tops:
54         if isinstance(v,DecisionVariable):
55             so += [p for p in v.parents if p not in so]
56             so.append(v)
57     so += [v for v in tops if v not in so]
58     return so

```

```

decnNetworks.py — (continued)
60 def show(self):
61     plt.ion() # interactive
62     ax = plt.figure().gca()
63     ax.set_axis_off()
64     plt.title(self.title)

```

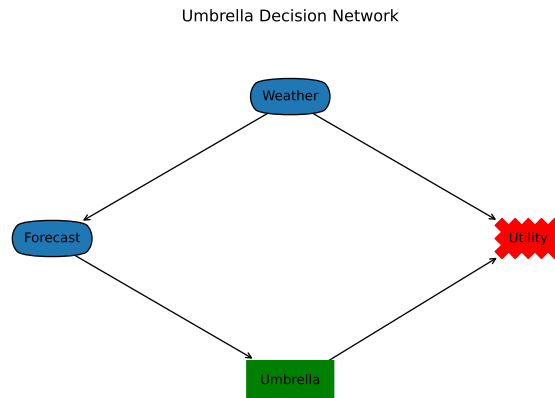


Figure 12.1: The umbrella decision network

```

65     for par in self.utility_factor.variables:
66         ax.annotate("Utility", par.position,
67                     xytext=self.utility_factor.position,
68                     arrowprops={'arrowstyle': '<-'}, bbox=dict(boxstyle="sawtooth,pad=1",
69                     ha='center')
69     for var in reversed(self.topological_sort()):
70         if isinstance(var, DecisionVariable):
71             bbox = dict(boxstyle="square,pad=1.0", color="green")
72         else:
73             bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5")
74         if self.var2parents[var]:
75             for par in self.var2parents[var]:
76                 ax.annotate(var.name, par.position, xytext=var.position,
77                             arrowprops={'arrowstyle': '<-'}, bbox=bbox,
78                             ha='center')
79         else:
80             x,y = var.position
81             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)
83 Weather = Variable("Weather", ["NoRain", "Rain"], position=(0.5,0.8))
84 Forecast = Variable("Forecast", ["Sunny", "Cloudy", "Rainy"],
85                      position=(0,0.4))
85 # Each variant uses one of the following:

```

```

86 | Umbrella = DecisionVariable("Umbrella", ["Take", "Leave"], {Forecast},
    |     position=(0.5,0))
87
88 | p_weather = Prob(Weather, [], [0.7, 0.3])
89 | p_forecast = Prob(Forecast, [Weather], [[0.7, 0.2, 0.1], [0.15, 0.25,
    |     0.6]])
90 | umb_utility = UtilityTable([Weather, Umbrella], [[20, 100], [70, 0]],
    |     position=(1,0.4))
91
92 | umbrella_dn = DecisionNetwork("Umbrella Decision Network",
    |     {Weather, Forecast, Umbrella},
93 |     {p_weather, p_forecast, umb_utility})
94

```

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) -----
96 | Umbrella2p = DecisionVariable("Umbrella", ["Take", "Leave"], {Forecast,
    |     Weather}, position=(0.5,0))
97 | umb_utility2p = UtilityTable([Weather, Umbrella2p], [[20, 100], [70, 0]],
    |     position=(1,0.4))
98 | umbrella_dn2p = DecisionNetwork("Umbrella Decision Network (extra arc)",
    |     {Weather, Forecast, Umbrella2p},
99 |     {p_weather, p_forecast, umb_utility2p})
100

```

Fire Decision Network

The fire decision network of Figure 12.2 (showing the result of `fire_dn.show()`) is represented as:

```

----- decnNetworks.py --- (continued) -----
102 | boolean = [False, True]
103 | Alarm = Variable("Alarm", boolean, position=(0.25,0.633))
104 | Fire = Variable("Fire", boolean, position=(0.5,0.9))
105 | Leaving = Variable("Leaving", boolean, position=(0.25,0.366))
106 | Report = Variable("Report", boolean, position=(0.25,0.1))
107 | Smoke = Variable("Smoke", boolean, position=(0.75,0.633))
108 | Tamper = Variable("Tamper", boolean, position=(0,0.9))
109
110 | See_Sm = Variable("See_Sm", boolean, position=(0.75,0.366) )
111 | Chk_Sm = DecisionVariable("Chk_Sm", boolean, {Report}, position=(0.5,
    |     0.366))
112 | Call = DecisionVariable("Call", boolean,{See_Sm,Chk_Sm,Report},
    |     position=(0.75,0.1))
113
114 | f_ta = Prob(Tamper,[],[0.98,0.02])
115 | f-fi = Prob(Fire,[],[0.99,0.01])
116 | f-sm = Prob(Smoke,[Fire],[[0.99,0.01],[0.1,0.9]])
117 | f-al = Prob(Alarm,[Fire,Tamper],[[0.9999, 0.0001], [0.15, 0.85]], [[0.01,
    |     0.99], [0.5, 0.5]])

```

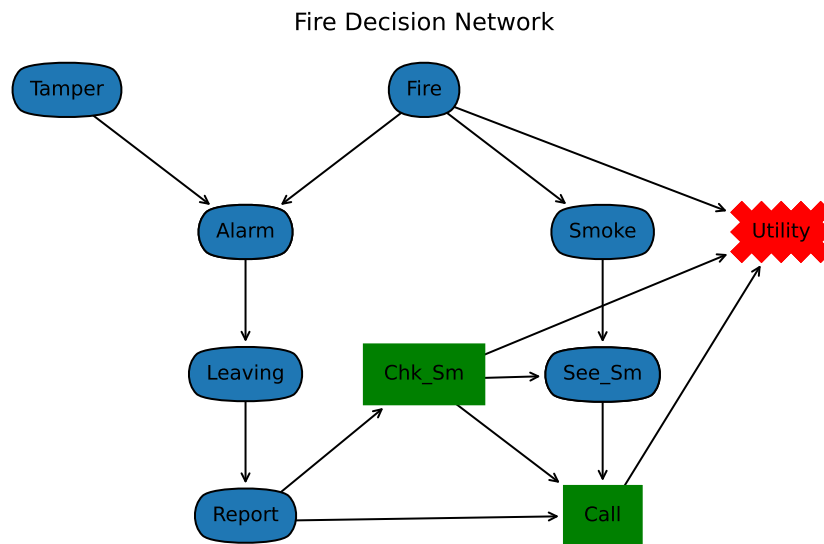


Figure 12.2: Fire Decision Network

```

118 f_lv = Prob(Leaving,[Alarm],[[0.999, 0.001], [0.12, 0.88]])
119 f_re = Prob(Report,[Leaving],[[0.99, 0.01], [0.25, 0.75]])
120 f_ss = Prob(See_Sm,[Chk_Sm,Smoke],[[1,0],[1,0]],[[1,0],[0,1]])
121
122 ut =
    UtilityTable([Chk_Sm,Fire,Call],[[0,-200],[-5000,-200]],[[-20,-220],[-5020,-220]],
    position=(1,0.633))
123
124 fire_dn = DecisionNetwork("Fire Decision Network",
125     {Tamper,Fire,Alarm,Leaving,Smoke,Call,See_Sm,Chk_Sm,Report},
126     {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)
128 grades = ['A','B','C','F']
129 Watched = Variable("Watched", boolean, position=(0,0.9))
130 Caught1 = Variable("Caught1", boolean, position=(0.2,0.7))
131 Caught2 = Variable("Caught2", boolean, position=(0.6,0.7))

```

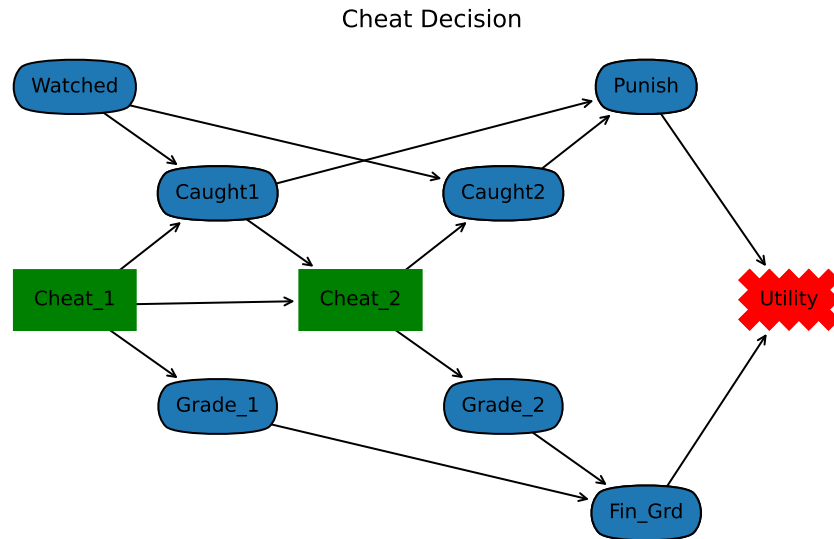


Figure 12.3: Cheating Decision Network

```

132 Punish = Variable("Punish", ["None","Suspension","Recorded"],
    position=(0.8,0.9))
133 Grade_1 = Variable("Grade_1", grades, position=(0.2,0.3))
134 Grade_2 = Variable("Grade_2", grades, position=(0.6,0.3))
135 Fin_Grd = Variable("Fin_Grd", grades, position=(0.8,0.1))
136 Cheat_1 = DecisionVariable("Cheat_1", boolean, set(), position=(0,0.5))
    #no parents
137 Cheat_2 = DecisionVariable("Cheat_2", boolean, {Cheat_1,Caught1},
    position=(0.4,0.5))
138
139 p_wa = Prob(Watched,[],[0.7, 0.3])
140 p_cc1 = Prob(Caught1,[Watched,Cheat_1],[[1.0, 0.0], [0.9, 0.1]], [[1.0,
    0.0], [0.5, 0.5]])
141 p_cc2 = Prob(Caught2,[Watched,Cheat_2],[[1.0, 0.0], [0.9, 0.1]], [[1.0,
    0.0], [0.5, 0.5]])
142 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]])
143 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}])
144 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}])
145 p_fg = Prob(Fin_Grd,[Grade_1,Grade_2],
146     {'A':{'A':1.0, 'B':0.0, 'C': 0.0, 'D':0.0},
147     'B': {'A':0.5, 'B':0.5, 'C': 0.0, 'D':0.0},

```

```

148         'C': {'A': 0.25, 'B': 0.5, 'C': 0.25, 'D': 0.0},
149         'D': {'A': 0.25, 'B': 0.25, 'C': 0.25, 'D': 0.25}},
150     'B': {'A': 0.5, 'B': 0.5, 'C': 0.0, 'D': 0.0},
151         'B': {'A': 0.0, 'B': 1, 'C': 0.0, 'D': 0.0},
152         'C': {'A': 0.0, 'B': 0.5, 'C': 0.5, 'D': 0.0},
153         'D': {'A': 0.0, 'B': 0.25, 'C': 0.5, 'D': 0.25}},
154     'C': {'A': 0.25, 'B': 0.5, 'C': 0.25, 'D': 0.0},
155         'B': {'A': 0.0, 'B': 0.5, 'C': 0.5, 'D': 0.0},
156         'C': {'A': 0.0, 'B': 0.0, 'C': 1, 'D': 0.0},
157         'D': {'A': 0.0, 'B': 0.0, 'C': 0.5, 'D': 0.5}},
158     'D': {'A': 0.25, 'B': 0.25, 'C': 0.25, 'D': 0.25},
159         'B': {'A': 0.0, 'B': 0.25, 'C': 0.5, 'D': 0.25},
160         'C': {'A': 0.0, 'B': 0.0, 'C': 0.5, 'D': 0.5},
161         'D': {'A': 0.0, 'B': 0.0, 'C': 0, 'D': 1.0}}})
162
163 utc = UtilityTable([Punish, Fin_Grd], {'None': {'A': 100, 'B': 90, 'C': 70,
164         'D': 50},
165         'Suspension': {'A': 40, 'B': 20, 'C': 10,
166         'D': 0},
167         'Recorded': {'A': 70, 'B': 60, 'C': 40,
168         'D': 20}}, position=(1, 0.5))
169
170 cheating_dn = DecisionNetwork("Cheating Decision Network",
171     {Punish, Caught2, Watched, Fin_Grd, Grade_2, Grade_1, Cheat_2, Caught1, Cheat_1},
172     {p_wa, p_cc1, p_cc2, p_pun, p_gr1,
173      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) -----
171 S0 = Variable('S0', boolean, position=(0, 0.5))
172 D0 = DecisionVariable('D0', boolean, {S0}, position=(1/7, 0.1))
173 S1 = Variable('S1', boolean, position=(2/7, 0.5))
174 D1 = DecisionVariable('D1', boolean, {S1}, position=(3/7, 0.1))
175 S2 = Variable('S2', boolean, position=(4/7, 0.5))
176 D2 = DecisionVariable('D2', boolean, {S2}, position=(5/7, 0.1))
177 S3 = Variable('S3', boolean, position=(6/7, 0.5))
178
179 p_s0 = Prob(S0, [], [0.5, 0.5])
180 tr = [[[0.1, 0.9], [0.9, 0.1]], [[0.2, 0.8], [0.8, 0.2]]] # 0 is flip, 1
    is keep value
181 p_s1 = Prob(S1, [D0, S0], tr)
182 p_s2 = Prob(S2, [D1, S1], tr)
183 p_s3 = Prob(S3, [D2, S2], tr)

```

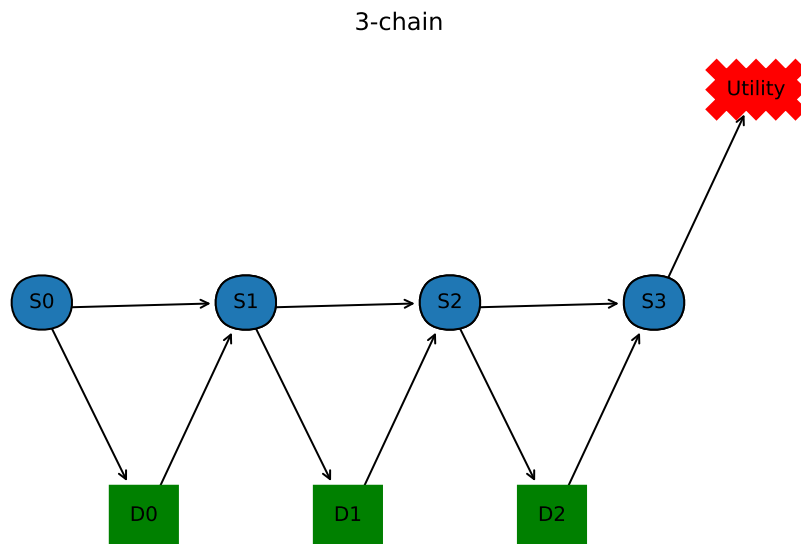


Figure 12.4: A decision network that is a chain of 3 decisions

```

184 |
185 | ch3U = UtilityTable([S3],[0,1], position=(7/7,0.9))
186 |
187 | ch3 = DecisionNetwork("3-chain",
188 |     {S0,D0,S1,D1,S2,D2,S3},{p_s0,p_s1,p_s2,p_s3,ch3U})
189 | #rc3 = RC_DN(ch3)
190 | #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)
192 | import math
193 | from probGraphicalModels import GraphicalModel, InferenceMethod
194 | from probFactors import Factor
195 | from probRC import connected_components
196 |
197 | class RC_DN(InferenceMethod):
198 |     """The class that queries graphical models using recursive conditioning
199 |

```



```

200     gm is graphical model to query
201     """
202
203     def __init__(self, gm=None):
204         self.gm = gm
205         self.cache = {(frozenset(), frozenset()):1}
206         ## self.max_display_level = 3
207
208     def optimize(self, split_order=None):
209         """computes expected utility, and creates optimal decision
210         functions, where
211         elim_order is a list of the non-observed non-query variables in gm
212         """
213         if split_order == None:
214             split_order = self.gm.split_order()
215             self.opt_policy = {}
216             return self.rc({}, self.gm.factors, split_order)

```

The following is the simplest search-based algorithm. It is exponential in the number of variables, so is not very useful. However, it is simple, and 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)
217     def rc0(self, context, factors, split_order):
218         """simplest search algorithm"""
219         self.display(2, "calling rc0,", (context, factors), "with
220             S0", split_order)
221         if not factors:
222             return 1
223         elif to_eval := {fac for fac in factors if
224             fac.can_evaluate(context)}:
225             self.display(3, "rc0 evaluating factors", to_eval)
226             val = math.prod(fac.get_value(context) for fac in to_eval)
227             return val * self.rc0(context, factors-to_eval, split_order)
228         else:
229             var = split_order[0]
230             self.display(3, "rc0 branching on", var)
231             if isinstance(var, DecisionVariable):
232                 assert set(context) <= set(var.parents), f"cannot optimize
233                     {var} in context {context}"
234                 maxres = -math.inf
235                 for val in var.domain:
236                     self.display(3, "In rc0, branching on", var, "=", val)
237                     newres = self.rc0({var:val}|context, factors,
238                         split_order[1:])
239                     if newres > maxres:
240                         maxres = newres
241                         theval = val
242                 self.opt_policy[frozenset(context.items())] = (var, theval)

```

```

239         return maxres
240     else:
241         total = 0
242         for val in var.domain:
243             total += self.rc0({var:val}|context, factors,
244                             split_order[1:])
245         self.display(3, "rc0 branching on", var,"returning", total)
246     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 198).

```

decnNetworks.py — (continued)
247 def rc(self, context, factors, split_order):
248     """ returns the number \sum_{split_order} \prod_{factors} given
249         assignments in context
250     context is a variable:value dictionary
251     factors is a set of factors
252     split_order is a list of variables in factors that are not in
253         context
254     """
255     self.display(3,"calling rc,", (context,factors))
256     ce = (frozenset(context.items()), frozenset(factors)) # key for the
257         cache entry
258     if ce in self.cache:
259         self.display(2,"rc cache lookup", (context,factors))
260         return self.cache[ce]
261     # if not factors: # no factors; needed if you don't have forgetting
262     # and caching
263     # return 1
264     elif vars_not_in_factors := {var for var in context
265                                 if not any(var in fac.variables for
266                                             fac in factors)}:
267         # forget variables not in any factor
268         self.display(3,"rc forgetting variables", vars_not_in_factors)
269         return self.rc({key:val for (key,val) in context.items()
270                         if key not in vars_not_in_factors},
271                       factors, split_order)
272     elif to_eval := {fac for fac in factors if
273                     fac.can_evaluate(context)}:
274         # evaluate factors when all variables are assigned
275         self.display(3,"rc evaluating factors", to_eval)
276         val = math.prod(fac.get_value(context) for fac in to_eval)
277         if val == 0:
278             return 0
279         else:
280             return val * self.rc(context, {fac for fac in factors if fac
281                                         not in to_eval}, split_order)
282     elif len(comp := connected_components(context, factors,
283                                         split_order)) > 1:

```

```

276         # there are disconnected components
277         self.display(2,"splitting into connected components",comp)
278         return(math.prod(self.rc(context,f,eo) for (f,eo) in comp))
279     else:
280         assert split_order, f"split_order empty rc({context},{factors})"
281         var = split_order[0]
282         self.display(3, "rc branching on", var)
283         if isinstance(var,DecisionVariable):
284             assert set(context) <= set(var.parents), f"cannot optimize
                {var} in context {context}"
285             maxres = -math.inf
286             for val in var.domain:
287                 self.display(3,"In rc, branching on",var,"=",val)
288                 newres = self.rc({var:val}|context, factors,
                    split_order[1:])
289                 if newres > maxres:
290                     maxres = newres
291                     theval = val
292                 self.opt_policy[frozenset(context.items())] = (var,theval)
293                 self.cache[ce] = maxres
294             return maxres
295         else:
296             total = 0
297             for val in var.domain:
298                 total += self.rc({var:val}|context, factors,
                    split_order[1:])
299             self.display(3, "rc branching on", var,"returning", total)
300             self.cache[ce] = total
301             return total

```

Here is how to run the optimize the example decision networks:

```

decnNetworks.py — (continued)
303 # Umbrella decision network
304 #urc = RC_DN(umberella_dn)
305 #urc.optimize()
306 #urc.opt_policy
307
308 #rc_fire = RC_DN(fire_dn)
309 #rc_fire.optimize()
310 #rc_fire.opt_policy
311
312 #rc_cheat = RC_DN(cheating_dn)
313 #rc_cheat.optimize()
314 #rc_cheat.opt_policy
315
316 #rc_ch3 = RC_DN(ch3)
317 #rc_ch3.optimize()
318 #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.)

```

320 from probVE import VE
321
322 class VE_DN(VE):
323     """Variable Elimination for Decision Networks"""
324     def __init__(self,dn=None):
325         """dn is a decision network"""
326         VE.__init__(self,dn)
327         self.dn = dn
328
329     def optimize(self,elim_order=None,obs={}):
330         if elim_order == None:
331             elim_order = reversed(self.gm.split_order())
332         policy = []
333         proj_factors = [self.project_observations(fac,obs)
334                         for fac in self.dn.factors]
335         for v in elim_order:
336             if isinstance(v,DecisionVariable):
337                 to_max = [fac for fac in proj_factors
338                           if v in fac.variables and set(fac.variables) <=
339                             v.all_vars]
340                 assert len(to_max)==1, "illegal variable order
341                    "+str(elim_order)+" at "+str(v)
342                 newFac = FactorMax(v, to_max[0])
343                 policy.append(newFac.decision_fun)
344                 proj_factors = [fac for fac in proj_factors if fac is not
345                               to_max[0]]+[newFac]
346                 self.display(2,"maximizing",v,"resulting
347                   factor",newFac.brief() )
348                 self.display(3,newFac)
349             else:
350                 proj_factors = self.eliminate_var(proj_factors, v)
351         assert len(proj_factors)==1,"Should there be only one element of
352           proj_factors?"
353         value = proj_factors[0].get_value({})
354         return value,policy

```

```

351 class FactorMax(Factor):
352     """A factor obtained by maximizing a variable in a factor.
353     Also builds a decision_function. This is based on FactorSum.
354     """

```

```

355
356     def __init__(self, dvar, factor):
357         """dvar is a decision variable.
358         factor is a factor that contains dvar and only parents of dvar
359         """
360         self.dvar = dvar
361         self.factor = factor
362         vars = [v for v in factor.variables if v is not dvar]
363         Factor.__init__(self, vars)
364         self.values = [None]*self.size
365         self.decision_fun = FactorDF(dvar, vars, [None]*self.size)
366
367     def get_value(self, assignment):
368         """lazy implementation: if saved, return saved value, else compute
369         it"""
370         index = self.assignment_to_index(assignment)
371         if self.values[index]:
372             return self.values[index]
373         else:
374             max_val = float("-inf") # -infinity
375             new_asst = assignment.copy()
376             for elt in self.dvar.domain:
377                 new_asst[self.dvar] = elt
378                 fac_val = self.factor.get_value(new_asst)
379                 if fac_val > max_val:
380                     max_val = fac_val
381                     best_elt = elt
382             self.values[index] = max_val
383             self.decision_fun.values[index] = best_elt
384         return max_val

```

A decision function is a stored factor.

```

385     class FactorDF(TabFactor):
386         """A decision function"""
387         def __init__(self, dvar, vars, values):
388             TabStored.__init__(self, vars, values)
389             self.dvar = dvar
390             self.name = str(dvar) # Used in printing

```

Here are some example queries:

```

392 # Example queries:
393 # v,p = VE_DN(fire_dn).optimize(); print(v)
394 # for df in p: print(df, "\n")
395
396 # VE_DN.max_display_level = 3 # if you want to show lots of detail
397 # v,p = VE_DN(cheating_dn).optimize(); print(v)
398 # 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_____
11 import random
12 import matplotlib.pyplot as plt
13 from matplotlib.widgets import Button, CheckButtons, TextBox
14 from display import Displayable
15 from utilities import argmaxd
16
17 class MDP(Displayable):
18     """A Markov Decision Process. Must define:
19     self.states the set (or list) of states
20     self.actions the set (or list) of actions
21     self.discount a real-valued discount
22     """
23
24     def __init__(self, states, actions, discount, init=0):
25         self.states = states
26         self.actions = actions
27         self.discount = discount
28         self.initv = self.v = {s:init for s in self.states}
29         self.initq = self.q = {s: {a: init for a in self.actions} for s in
30                                 self.states}
31
32     def P(self,s,a):
33         """Transition probability function
34         returns a dictionary of {s1:p1} such that P(s1 | s,a)=p1. Other
35         probabilities are zero.
36         """
37         raise NotImplementedError("P") # abstract method
38
39     def R(self,s,a):
40         """Reward function R(s,a)
41         returns the expected reward for doing a in state s.
42         """
43         raise NotImplementedError("R") # abstract method

```

Two state partying example (Example 9.27 in Poole and Mackworth [2017]):

```

_____mdpExamples.py — MDP Examples_____
11 from mdpProblem import MDP, GridMDP
12 import matplotlib.pyplot as plt
13
14 class party(MDP):
15     """Simple 2-state, 2-Action Partying MDP Example"""
16     def __init__(self, discount=0.9):
17         states = {'healthy', 'sick'}

```

```

18     actions = {'relax', 'party'}
19     MDP.__init__(self, states, actions, discount)
20
21     def R(self,s,a):
22         "R(s,a)"
23         return { 'healthy': {'relax': 7, 'party': 10},
24                 'sick':    {'relax': 0, 'party': 2 }}[s][a]
25
26     def P(self,s,a):
27         "returns a dictionary of {s1:p1} such that P(s1 | s,a)=p1. Other
28         probabilities are zero."
29         phealthy = { # P('healthy' | s, a)
30                     'healthy': {'relax': 0.95, 'party': 0.7},
31                     'sick':    {'relax': 0.5, 'party': 0.1 }}[s][a]
32         return {'healthy':phealthy, 'sick':1-phealthy}

```

The 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)

```

34 class MDPTiny(GridMDP):
35     def __init__(self, discount=0.9):
36         actions = ['right', 'upC', 'left', 'upR']
37         self.x_dim = 2 # x-dimension
38         self.y_dim = 3
39         states = [(x,y) for x in range(self.x_dim) for y in
40                     range(self.y_dim)]
41         # for GridMDP
42         self.xoff = {'right':0.25, 'upC':0, 'left':-0.25, 'upR':0}
43         self.yoff = {'right':0, 'upC':-0.25, 'left':0, 'upR':0.25}
44         GridMDP.__init__(self, states, actions, discount)
45
46     def P(self,s,a):
47         """return a dictionary of {s1:p1} if P(s1 | s,a)=p1. Other
48         probabilities are zero.
49         """
50         (x,y) = s
51         if a == 'right':
52             return {(1,y):1}
53         elif a == 'upC':
54             return {(x,min(y+1,2)):1}
55         elif a == 'left':
56             if (x,y) == (0,2): return {(0,0):1}
57             else: return {(0,y): 1}
58         elif a == 'upR':
59             if x==0:
60                 if y<2: return {(x,y):0.1, (x+1,y):0.1, (x,y+1):0.8}

```

```

59         else: # at (0,2)
60             return {(0,0):0.1, (1,2): 0.1, (0,2): 0.8}
61     elif y < 2: # x==1
62         return {(0,y):0.1, (1,y):0.1, (1,y+1):0.8}
63     else: # at (1,2)
64         return {(0,2):0.1, (1,2): 0.9}
65
66     def R(self,s,a):
67         (x,y) = s
68         if a == 'right':
69             return [0,-1][x]
70         elif a == 'upC':
71             return [-1,-1,-2][y]
72         elif a == 'left':
73             if x==0:
74                 return [-1, -100, 10][y]
75             else: return 0
76         elif a == 'upR':
77             return [[-0.1, -10, 0.2],[-0.1, -0.1, -0.9]][x][y]
78             # 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)_____
80 class grid(GridMDP):
81     """ x_dim * y_dim grid with rewarding states"""
82     def __init__(self, discount=0.9, x_dim=10, y_dim=10):
83         self.x_dim = x_dim # size in x-direction
84         self.y_dim = y_dim # size in y-direction
85         actions = ['up', 'down', 'right', 'left']
86         states = [(x,y) for x in range(y_dim) for y in range(y_dim)]
87         self.rewarding_states = {(3,2):-10, (3,5):-5, (8,2):10, (7,7):3}
88         self.fling_states = {(8,2), (7,7)}
89         self.xoff = {'right':0.25, 'up':0, 'left':-0.25, 'down':0}
90         self.yoff = {'right':0, 'up':0.25, 'left':0, 'down':-0.25}
91         GridMDP.__init__(self, states, actions, discount)
92
93     def intended_next(self,s,a):
94         """returns the next state in the direction a.
95         This is where the agent will end up if to goes in its
96             intended_direction
97             (which it does with probability 0.7).
98         """
99         (x,y) = s
100         if a=='up':
101             return (x, y+1 if y+1 < self.y_dim else y)
102         if a=='down':
103             return (x, y-1 if y > 0 else y)
104         if a=='right':

```



```

104         return (x+1 if x+1 < self.x_dim else x,y)
105     if a=='left':
106         return (x-1 if x > 0 else x,y)
107
108     def P(self,s,a):
109         """return a dictionary of {s1:p1} if P(s1 | s,a)=p1. Other
110         probabilities are zero.
111         Corners are tricky because different actions result in same state.
112         """
113         if s in self.fling_states:
114             return {(0,0): 0.25, (self.x_dim-1,0):0.25,
115                     (0,self.y_dim-1):0.25, (self.x_dim-1,self.y_dim-1):0.25}
116         res = dict()
117         for ai in self.actions:
118             s1 = self.intended_next(s,ai)
119             ps1 = 0.7 if ai==a else 0.1
120             if s1 in res: # occurs in corners
121                 res[s1] += ps1
122             else:
123                 res[s1] = ps1
124         return res
125
126     def R(self,s,a):
127         if s in self.rewarding_states:
128             return self.rewarding_states[s]
129         else:
130             (x,y) = s
131             rew = 0
132             # rewards from crashing:
133             if y==0: ## on bottom.
134                 rew += -0.7 if a == 'down' else -0.1
135             if y==self.y_dim-1: ## on top.
136                 rew += -0.7 if a == 'up' else -0.1
137             if x==0: ## on left
138                 rew += -0.7 if a == 'left' else -0.1
139             if x==self.x_dim-1: ## on right.
140                 rew += -0.7 if a == 'right' else -0.1
141         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) _

```

43     def vi(self, n):
44         """carries out n iterations of value iteration, updating value
           function self.v
45         Returns a Q-function, value function, policy
46         """
47         self.display(3,"calling vi")
48         assert n>0,"You must carry out at least one iteration of vi.
           n="+str(n)
49         #v = v0 if v0 is not None else {s:0 for s in self.states}
50         for i in range(n):
51             self.q = {s: {a: self.R(s,a)+self.discount*sum(p1*self.v[s1]
52                                                         for (s1,p1) in
53                                                         self.P(s,a).items())
54                     for a in self.actions}
55             for s in self.states}
56             self.v = {s: max(self.q[s][a] for a in self.actions)
57                     for s in self.states}
58             self.pi = {s: argmaxd(self.q[s])
59                     for s in self.states}
59         return self.q, self.v, self.pi

```

The following shows how this can be used.

```

_____mdpExamples.py — (continued)_____
141 ## Testing value iteration
142 # Try the following:
143 # pt = party(discount=0.9)
144 # pt.vi(1)
145 # pt.vi(100)
146 # party(discount=0.99).vi(100)
147 # party(discount=0.4).vi(100)
148
149 # gr = grid(discount=0.9)
150 # gr.show()
151 # q,v,pi = gr.vi(100)
152 # 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)_____
61 class GridMDP(MDP):
62     def __init__(self, states, actions, discount):
63         MDP.__init__(self, states, actions, discount)
64
65     def show(self):
66         plt.ion() # interactive
67         fig,(self.ax) = plt.subplots()

```

```

68     plt.subplots_adjust(bottom=0.2)
69     stepB = Button(plt.axes([0.8,0.05,0.1,0.075]), "step")
70     stepB.on_clicked(self.on_step)
71     resetB = Button(plt.axes([0.65,0.05,0.1,0.075]), "reset")
72     resetB.on_clicked(self.on_reset)
73     self.qcheck = CheckButtons(plt.axes([0.2,0.05,0.35,0.075]),
74                                ["show q-values","show policy"])
75     self.qcheck.on_clicked(self.show_vals)
76     self.font_box = TextBox(plt.axes([0.1,0.05,0.05,0.075]),"Font:",
77                              textalignment="center")
78     self.font_box.on_submit(self.set_font_size)
79     self.font_box.set_val(str(plt.rcParams['font.size']))
80     self.show_vals(None)
81     plt.show()
82
83     def set_font_size(self, s):
84         plt.rcParams.update({'font.size': eval(s)})
85         plt.draw()
86
87     def show_vals(self,event):
88         self.ax.cla()
89         array = [[self.v[(x,y)] for x in range(self.x_dim)
90                  for y in range(self.y_dim)]
91                  [x-0.5 for x in range(self.x_dim+1)],
92                  [x-0.5 for x in range(self.y_dim+1)],
93                  array, edgecolors='black',cmap='summer')
94
95         # for cmap see
96         # https://matplotlib.org/stable/tutorials/colors/colormaps.html
97         if self.qcheck.get_status()[1]: # "show policy"
98             for (x,y) in self.q:
99                 maxv = max(self.q[(x,y)][a] for a in self.actions)
100                 for a in self.actions:
101                     if self.q[(x,y)][a] == maxv:
102                         # draw arrow in appropriate direction
103                         self.ax.arrow(x,y,self.xoff[a]*2,self.yoff[a]*2,
104                                       color='red',width=0.05, head_width=0.2,
105                                       length_includes_head=True)
106         if self.qcheck.get_status()[0]: # "show q-values"
107             self.show_q(event)
108         else:
109             self.show_v(event)
110         self.ax.set_xticks(range(self.x_dim))
111         self.ax.set_xticklabels(range(self.x_dim))
112         self.ax.set_yticks(range(self.y_dim))
113         self.ax.set_yticklabels(range(self.y_dim))
114         plt.draw()
115
116     def on_step(self,event):
117         self.vi(1)
118         self.show_vals(event)

```

```

115
116     def show_v(self, event):
117         """show values"""
118         for (x,y) in self.v:
119             self.ax.text(x,y, "{val:.2f}".format(val=self.v[(x,y)]), ha='center')
120
121     def show_q(self, event):
122         """show q-values"""
123         for (x,y) in self.q:
124             for a in self.actions:
125                 self.ax.text(x+self.xoff[a], y+self.yoff[a],
126                             "{val:.2f}".format(val=self.q[(x,y)][a]), ha='center')
127
128     def on_reset(self, event):
129         self.v = self.initv
130         self.q = self.initq
131         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) _

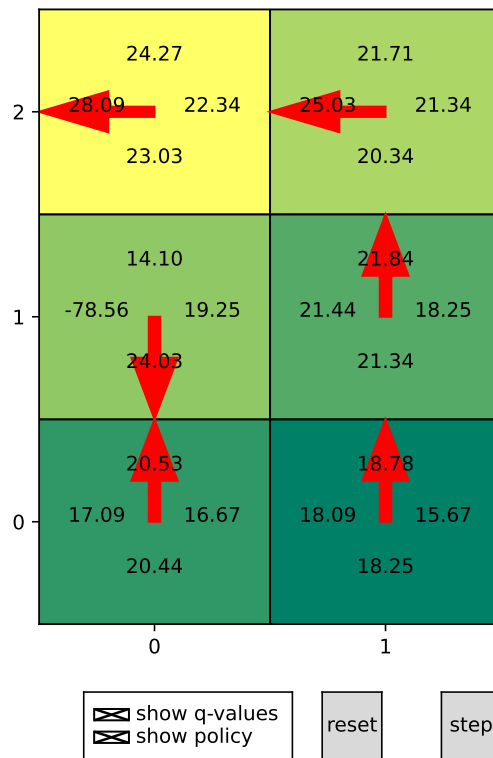


Figure 12.5: Interface for tiny example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The leftmost number is the for the left action; the rightmost number is for the right action; the upper most is for the *upR* (up-risky) action and 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`

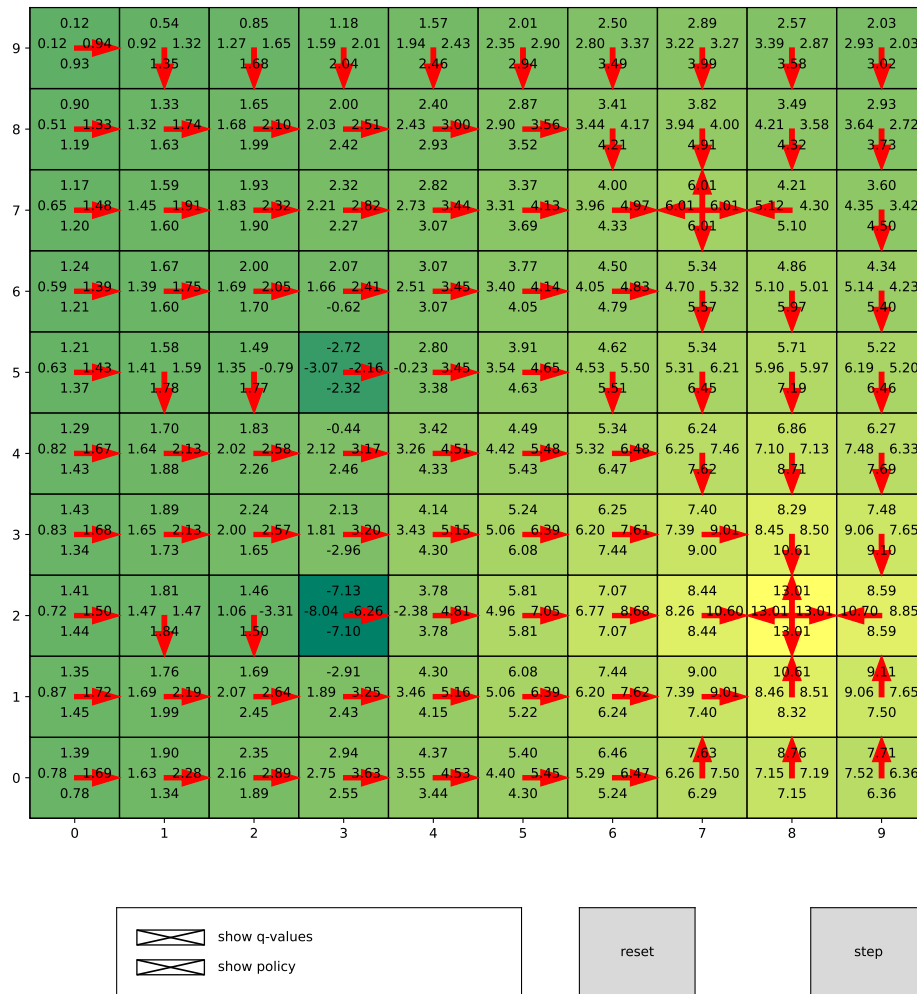


Figure 12.6: Interface for grid example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The leftmost number is the for the left action; the rightmost number is for the right action; the upper most is for the up action and the lowest number is for the down action. The arrow points to the action(s) with the maximum Q-value. From `grid(discount=0.9).show()`

```

133     def avi(self,n):
134         states = list(self.states)
135         actions = list(self.actions)
136         for i in range(n):
137             s = random.choice(states)
138             a = random.choice(actions)
139             self.q[s][a] = (self.R(s,a) + self.discount *
140                             sum(p1 * max(self.q[s1][a1]
141                                           for a1 in self.actions)
142                                for (s1,p1) in self.P(s,a).items()))
143         return Q

```

The following shows how avi can be used.

```

_____mdpExamples.py — (continued) _____
155 ## Testing asynchronous value iteration
156 # Try the following:
157 # pt = party(discount=0.9)
158 # pt.avi(10)
159 # pt.vi(1000)
160
161 # gr = grid(discount=0.9)
162 # q = gr.avi(100000)
163 # 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.

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 _____
11 import random
12 import math
13 from display import Displayable
14 from agents import Agent, Environment
15 from utilities import pick_from_dist, argmaxe, argmaxd, flip
16
17 class RL_env(Environment):
```

```

18     def __init__(self, name, actions, state):
19         """creates an environment given name, list of actions, and initial
           state"""
20         self.name = name          # the role for an agent
21         self.actions = actions    # list of all actions
22         self.state = state        # initial state
23
24         # must implement do(action)->(reward,state)

```

rlProblem.py — (continued)

```

26
27     class RL_agent(Agent):
28         """An RL_Agent
29         has percepts (s, r) for some state s and real reward r
30         """
31         def __init__(self, actions):
32             self.actions = actions
33
34         def initial_action(self, env_state):
35             """return the initial action, and remember the state and action
36             Act randomly initially
37             Could be overridden to initialize data structures (as the agent now
38             knows about one state)
39             """
40             self.state = env_state
41             self.action = random.choice(self.actions)
42             return self.action
43
44         def select_action(self, reward, state):
45             """
46             Select the action given the reward and next state
47             Remember the action in self.action
48             This implements "Act randomly" and should be overridden!
49             """
50             self.action = random.choice(self.actions)
51             return self.action

```

This is similar to Simulate of Section 2.1, except it is initialized by `agent.initial_action(state)`.

rlProblem.py — (continued)

```

52     import matplotlib.pyplot as plt
53
54     class Simulate(Displayable):
55         """simulate the interaction between the agent and the environment
56         for n time steps.
57         Returns a pair of the agent state and the environment state.
58         """
59         def __init__(self, agent, environment):
60             self.agent = agent
61             self.env = environment
62             self.action = agent.initial_action(self.env.state)

```

```

63         self.reward_history = [] # for plotting
64
65     def go(self, n):
66         for i in range(n):
67             (reward,state) = self.env.do(self.action)
68             self.display(2,f"i={i} reward={reward}, state={state}")
69             self.reward_history.append(reward)
70             self.action = self.agent.select_action(reward,state)
71             self.display(2,f"    action={self.action}")
72         return self

```

The following plots the sum of rewards as a function of the step in a simulation.

```

                                rlProblem.py — (continued)
74     def plot(self, label=None, step_size=None, xscale='linear'):
75         """
76         plots the rewards history in the simulation
77         label is the label for the plot
78         step_size is the number of steps between each point plotted
79         xscale is 'log' or 'linear'
80
81         returns sum of rewards
82         """
83         if step_size is None: #for long simulations (> 999), only plot some
            points
84             step_size = max(1,len(self.reward_history)//500)
85         if label is None:
86             label = self.agent.method
87         plt.ion()
88         plt.xscale(xscale)
89         plt.xlabel("step")
90         plt.ylabel("Sum of rewards")
91         sum_history, sum_rewards = acc_rews(self.reward_history, step_size)
92         plt.plot(range(0,len(self.reward_history),step_size), sum_history,
93                 label=label)
94         plt.legend()
95         plt.draw()
96         return sum_rewards
97
98     def acc_rews(rews,step_size):
99         """returns the rolling sum of the values, sampled each step_size, and
100            the sum
101            """
102         acc = []
103         sumr = 0; i=0
104         for e in rews:
105             sumr += e
106             i += 1
107             if (i%step_size == 0): acc.append(sumr)
108         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 _____
11 from rlProblem import RL_env
12 class Healthy_env(RL_env):
13     def __init__(self):
14         RL_env.__init__(self, "Party Decision", ["party", "relax"],
15             "healthy")
16
17     def do(self, action):
18         """updates the state based on the agent doing action.
19         returns reward,state
20         """
21         if self.state=="healthy":
22             if action=="party":
23                 self.state = "healthy" if flip(0.7) else "sick"
24                 reward = 10
25             else: # action=="relax"
26                 self.state = "healthy" if flip(0.95) else "sick"
27                 reward = 7
28         else: # self.state=="sick"
29             if action=="party":
30                 self.state = "healthy" if flip(0.1) else "sick"
31                 reward = 2
32             else:
33                 self.state = "healthy" if flip(0.5) else "sick"
34                 reward = 0
35         return reward, self.state

```

13.1.1 Simulating an environment from an MDP

Given the definition for an MDP (page 262), *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) _____
109 class Env_from_MDP(RL_env):
110     def __init__(self, mdp):
111         initial_state = random.choice(mdp.states)
112         RL_env.__init__(self, "From MDP", mdp.actions, initial_state)
113         self.mdp = mdp
114
115     def do(self, action):

```

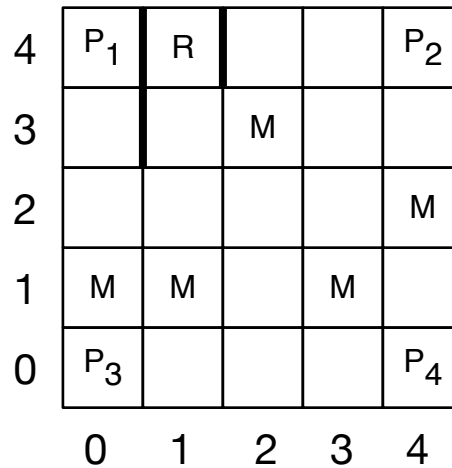


Figure 13.1: Monster game

```

116     """updates the state based on the agent doing action.
117     returns state,reward
118     """
119     reward = self.mdp.R(self.state,action)
120     self.state = pick_from_dist(self.mdp.P(self.state,action))
121     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)
36 import random
37 from utilities import flip
38 from rlProblem import RL_env
39
40 class Monster_game_env(RL_env):
41     xdim = 5
42     ydim = 5
43
44     vwalls = [(0,3), (0,4), (1,4)] # vertical walls right of these locations
45     hwalls = [] # not implemented
46     crashed_reward = -1
47
48     prize_locs = [(0,0), (0,4), (4,0), (4,4)]
49     prize_appears_prob = 0.3
50     prize_reward = 10
51
52     monster_locs = [(0,1), (1,1), (2,3), (3,1), (4,2)]

```

```

53 monster_appears_prob = 0.4
54 monster_reward_when_damaged = -10
55 repair_stations = [(1,4)]
56
57 actions = ["up","down","left","right"]
58
59 def __init__(self):
60     # State:
61     self.x = 2
62     self.y = 2
63     self.damaged = False
64     self.prize = None
65     # Statistics
66     self.number_steps = 0
67     self.accumulated_rewards = 0 # sum of rewards received
68     self.min_accumulated_rewards = 0
69     self.min_step = 0
70     self.zero_crossing = 0
71     RL_env.__init__(self, "Monster Game", self.actions, (self.x,
72         self.y, self.damaged, self.prize))
73     self.display(2,"","Step","Tot Rew","Ave Rew",sep="\t")
74
75 def do(self,action):
76     """updates the state based on the agent doing action.
77     returns reward,state
78     """
79     assert action in self.actions, f"Monster game, unknown action:
80         {action}"
81     reward = 0.0
82     # A prize can appear:
83     if self.prize is None and flip(self.prize_appears_prob):
84         self.prize = random.choice(self.prize_locs)
85     # Actions can be noisy
86     if flip(0.4):
87         actual_direction = random.choice(self.actions)
88     else:
89         actual_direction = action
90     # Modeling the actions given the actual direction
91     if actual_direction == "right":
92         if self.x==self.xdim-1 or (self.x,self.y) in self.vwalls:
93             reward += self.crashed_reward
94         else:
95             self.x += 1
96     elif actual_direction == "left":
97         if self.x==0 or (self.x-1,self.y) in self.vwalls:
98             reward += self.crashed_reward
99         else:
100             self.x += -1
101     elif actual_direction == "up":
102         if self.y==self.ydim-1:

```

```

101         reward += self.crashed_reward
102     else:
103         self.y += 1
104     elif actual_direction == "down":
105         if self.y==0:
106             reward += self.crashed_reward
107         else:
108             self.y += -1
109     else:
110         raise RuntimeError(f"unknown_direction: {actual_direction}")
111
112     # Monsters
113     if (self.x,self.y) in self.monster_locs and
114         flip(self.monster_appears_prob):
115         if self.damaged:
116             reward += self.monster_reward_when_damaged
117         else:
118             self.damaged = True
119     if (self.x,self.y) in self.repair_stations:
120         self.damaged = False
121
122     # Prizes
123     if (self.x,self.y) == self.prize:
124         reward += self.prize_reward
125         self.prize = None
126
127     # Statistics
128     self.number_steps += 1
129     self.accumulated_rewards += reward
130     if self.accumulated_rewards < self.min_accumulated_rewards:
131         self.min_accumulated_rewards = self.accumulated_rewards
132         self.min_step = self.number_steps
133     if self.accumulated_rewards>0 and reward>self.accumulated_rewards:
134         self.zero_crossing = self.number_steps
135     self.display(2,"",self.number_steps,self.accumulated_rewards,
136                 self.accumulated_rewards/self.number_steps,sep="\t")
137
138     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

```

11 import random
12 import math

```

```

13 from display import Displayable
14 from utilities import argmaxe, argmaxd, flip
15 from rlProblem import RL_agent, epsilon_greedy, ucb
16
17 class Q_learner(RL_agent):
18     """A Q-learning agent has
19     belief-state consisting of
20     state is the previous state (initialized by RL_agent
21     q is a {(state,action):value} dict
22     visits is a {(state,action):n} dict. n is how many times action was
23     done in state
24     acc_rewards is the accumulated reward
25     """

```

rlQLearner.py — (continued)

```

27 def __init__(self, role, actions, discount,
28             exploration_strategy=epsilon_greedy, es_kwargs={},
29             alpha_fun=lambda _:0.2,
30             Qinit=0, method="Q_learner"):
31     """
32     role is the role of the agent (e.g., in a game)
33     actions is the set of actions the agent can do
34     discount is the discount factor
35     exploration_strategy is the exploration function, default
36         "epsilon_greedy"
37     es_kwargs is extra arguments of exploration_strategy
38     alpha_fun is a function that computes alpha from the number of
39         visits
40     Qinit is the initial q-value
41     method gives the method used to implement the role (for plotting)
42     """
43     RL_agent.__init__(self, actions)
44     self.role = role
45     self.discount = discount
46     self.exploration_strategy = exploration_strategy
47     self.es_kwargs = es_kwargs
48     self.alpha_fun = alpha_fun
49     self.Qinit = Qinit
50     self.method = method
51     self.acc_rewards = 0
52     self.Q = {}
53     self.visits = {}

```

The initial action is a random action. It remembers the state, and initializes the data structures.

rlQLearner.py — (continued)

```

53 def initial_action(self, state):
54     """ Returns the initial action; selected at random
55     Initialize Data Structures

```



```

56     """
57     self.Q[state] = {act:self.Qinit for act in self.actions}
58     self.action = random.choice(self.actions)
59     self.visits[state] = {act:0 for act in self.actions}
60     self.state = state
61     self.display(2, f"Initial State: {state} Action {self.action}")
62     self.display(2,"s\ta\t\r\t's'\tQ")
63     return self.action
64
65 def select_action(self, reward, next_state):
66     """give reward and next state, select next action to be carried
        out"""
67     if next_state not in self.visits: # next state not seen before
68         self.Q[next_state] = {act:self.Qinit for act in self.actions}
69         self.visits[next_state] = {act:0 for act in self.actions}
70     self.visits[self.state][self.action] +=1
71     alpha = self.alpha_fun(self.visits[self.state][self.action])
72     self.Q[self.state][self.action] += alpha*(
73         reward
74         + self.discount * max(self.Q[next_state].values())
75         - self.Q[self.state][self.action])
76     self.display(2,self.state, self.action, reward, next_state,
77                 self.Q[self.state][self.action], sep='\t')
78     self.state = next_state
79     self.action = self.exploration_strategy(self.Q[next_state],
80                                             self.visits[next_state]**self.es_kwargs)
81     self.display(3,f"Agent {self.role} doing {self.action} in state
        {self.state}")
82     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

- Q_s is a $\{action : q_value\}$ dictionary for the current state
- V_s 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.

```

123 def epsilon_greedy(Qs, Vs={}, epsilon=0.1):
124     """select action given epsilon greedy
125     Qs is the {action:Q-value} dictionary for current state
126     Vs is ignored
127     """

```

```

128     if flip(epsilon):
129         return random.choice(list(Qs.keys())) # act randomly
130     else:
131         return argmaxd(Qs)
132
133 def ucb(Qs, Vs, c=1.4):
134     """select action given upper-confidence bound
135     Qs is the {action:Q-value} dictionary for current state
136     Vs is the {action:visits} dictionary for current state
137
138     0.01 is to prevent divide-by zero (could just be infinity)
139     """
140     Ns = sum(Vs.values())
141     ucb1 = {a:Qs[a]+c*math.sqrt(Ns/(0.01+Vs[a]))
142            for a in Qs.keys()}
143     action = argmaxd(ucb1)
144     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)_____
84 ##### TEST CASES #####
85 from rlProblem import Simulate, epsilon_greedy, ucb
86 from rlExamples import Healthy_env, Monster_game_env
87 from rlQLearner import Q_learner
88
89 env = Healthy_env()
90 # Some RL agents with different parameters:
91 ag = Q_learner(env.name, env.actions, 0.7)
92 ag_ucb = Q_learner(env.name, env.actions, 0.7, exploration_strategy = ucb,
93                   es_kwargs={'c':0.1}, method="ucb")
94 ag_opt = Q_learner(env.name, env.actions, 0.7, Qinit=100,
95                   method="optimistic" )
96 ag_exp_m = Q_learner(env.name, env.actions, 0.7,
97                   es_kwargs={'epsilon':0.5}, method="more explore")
98 ag_greedy = Q_learner(env.name, env.actions, 0.1, Qinit=100, method="disc
99                   0.1")
100
101 sim_ag = Simulate(ag, env)
102
103 # sim_ag.go(100)
104 # ag.Q # get the learned Q-values
105 # sim_ag.plot()

```

```

102 # Simulate(ag_uctb,env).go(100).plot()
103 # Simulate(ag_opt,env).go(100).plot()
104 # Simulate(ag_exp_m,env).go(100).plot()
105 # Simulate(ag_greedy,env).go(100).plot()
106
107
108 from mdpExamples import MDPTiny
109 from rlProblem import Env_from_MDP
110 envt = Env_from_MDP(MDPTiny())
111 agt = Q_learner(envt.name, envt.actions, 0.8)
112 #Simulate(agt, envt).go(1000).plot()
113
114 mon_env = Monster_game_env()
115 mag1 = Q_learner(mon_env.name, mon_env.actions,0.9)
116 #Simulate(mag1,mon_env).go(100000).plot()
117 mag_uctb = Q_learner(mon_env.name, mon_env.actions,0.9,exploration_strategy
118                 = ucb,es_kwargs={'c':0.1},method="UCB(0.1)")
119 #Simulate(mag_uctb,mon_env).go(100000).plot()
120
121 mag2 = Q_learner(mon_env.name, mon_env.actions,
122                 0.9,es_kwargs={'epsilon':0.2},alpha_fun=lambda
123                 k:1/k,method="alpha=1/k")
124 #Simulate(mag2,mon_env).go(100000).plot()
125 mag3 = Q_learner(mon_env.name, mon_env.actions, 0.9,alpha_fun=lambda
126                 k:10/(9+k),method="alpha=10/(9+k)")
127 #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_____
11 from rlQLearner import Q_learner
12 from utilities import flip
13 import random
14
15 class BoundedBuffer(object):
16     def __init__(self, buffer_size=1000):
17         self.buffer_size = buffer_size
18         self.buffer = [0]*buffer_size
19         self.number_added = 0
20
21     def add(self,experience):
22         if self.number_added < self.buffer_size:
23             self.buffer[self.number_added] = experience
24         else:
25             if flip(self.buffer_size/self.number_added):
26                 position = random.randrange(self.buffer_size)

```

```

27         self.buffer[position] = experience
28         self.number_added += 1
29
30     def get(self):
31         return self.buffer[random.randrange(min(self.number_added,
                                                    self.buffer_size))]

```

A Q_ER_Learner does Q-learning with experience replay. It only uses action replay after burn_in number of steps.

```

rlQExperienceReplay.py — (continued)
33 class Q_ER_learner(Q_learner):
34     def __init__(self, role, actions, discount,
35                  max_buffer_size=10000,
36                  num_updates_per_action=5, burn_in=1000,
37                  method="Q_ER_learner", **q_kwargs):
38         """Q-learner with experience replay
39         role is the role of the agent (e.g., in a game)
40         actions is the set of actions the agent can do
41         discount is the discount factor
42         max_buffer_size is the maximum number of past experiences that is
43             remembered
44         burn_in is the number of steps before using old experiences
45         num_updates_per_action is the number of q-updates for past
46             experiences per action
47         q_kwargs are any extra parameters for Q_learner
48         """
49         Q_learner.__init__(self, role, actions, discount, method=method,
50                             **q_kwargs)
51         self.experience_buffer = BoundedBuffer(max_buffer_size)
52         self.num_updates_per_action = num_updates_per_action
53         self.burn_in = burn_in
54
55     def select_action(self, reward, next_state):
56         """give reward and new state, select next action to be carried
57             out"""
58         self.experience_buffer.add((self.state, self.action, reward, next_state))
59         #remember experience
60         if next_state not in self.Q: # Q and visits are defined on the same
61             states
62             self.Q[next_state] = {act:self.Qinit for act in self.actions}
63             self.visits[next_state] = {act:0 for act in self.actions}
64         self.visits[self.state][self.action] +=1
65         alpha = self.alpha_fun(self.visits[self.state][self.action])
66         self.Q[self.state][self.action] += alpha*(
67             reward
68             + self.discount * max(self.Q[next_state].values())
69             - self.Q[self.state][self.action])
70         self.display(2, self.state, self.action, reward, next_state,
71                     self.Q[self.state][self.action], sep='\t')
72         self.state = next_state

```

```

67         # do some updates from experience buffer
68         if self.experience_buffer.number_added > self.burn_in:
69             for i in range(self.num_updates_per_action):
70                 (s,a,r,ns) = self.experience_buffer.get()
71                 self.visits[s][a] +=1 # is this correct?
72                 alpha = self.alpha_fun(self.visits[s][a])
73                 self.Q[s][a] += alpha * (r +
74                                         self.discount* max(self.Q[ns][na]
75                                                             for na in self.actions)
76                                         -self.Q[s][a] )
77             ### CHOOSE NEXT ACTION ###
78             self.action = self.exploration_strategy(self.Q[next_state],
79                                                    self.visits[next_state],**self.es_kwargs)
80             self.display(3,f"Agent {self.role} doing {self.action} in state
81                        {self.state}")
82             return self.action

```

rlQExperienceReplay.py — (continued)

```

83 from rlProblem import Simulate
84 from rlExamples import Monster_game_env
85 from rlQLearner import mag1, mag2, mag3
86
87 mon_env = Monster_game_env()
88 mag1ar = Q_ER_learner(mon_env.name, mon_env.actions,0.9,method="Q_ER")
89 # Simulate(mag1ar,mon_env).go(100000).plot()
90
91 mag3ar = Q_ER_learner(mon_env.name, mon_env.actions, 0.9, alpha_fun=lambda
92                        k:10/(9+k),method="Q_ER alpha=10/(9+k)")
93 # 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
11 import random
12 from rlProblem import RL_agent, Simulate, epsilon_greedy, ucb
13 from display import Displayable
14 from utilities import argmaxe, flip
15
16 class Model_based_reinforcement_learner(RL_agent):
17     """A Model-based reinforcement learner
18     """
19
20     def __init__(self, role, actions, discount,
21                 exploration_strategy=epsilon_greedy, es_kwargs={},
22                 Qinit=0,
23                 updates_per_step=10, method="MBR_learner"):
24         """role is the role of the agent (e.g., in a game)
25         actions is the list of actions the agent can do
26         discount is the discount factor
27         explore is the proportion of time the agent will explore
28         Qinit is the initial value of the Q's
29         updates_per_step is the number of AVI updates per action
30         label is the label for plotting
31         """
32         RL_agent.__init__(self, actions)
33         self.role = role
34         self.actions = actions
35         self.discount = discount
36         self.exploration_strategy = exploration_strategy
37         self.es_kwargs = es_kwargs
38         self.Qinit = Qinit
39         self.updates_per_step = updates_per_step
40         self.method = method
41
42
43 rlModelLearner.py — (continued)
42 def initial_action(self, state):
43     """ Returns the initial action; selected at random

```

```
44 Initialize Data Structures
```

```
45
46 """
47 self.state = state
48 self.T = {self.state: {a: {} for a in self.actions}}
49 self.visits = {self.state: {a: 0 for a in self.actions}}
50 self.Q = {self.state: {a: self.Qinit for a in self.actions}}
51 self.R = {self.state: {a: 0 for a in self.actions}}
52 self.states_list = [self.state] # list of states encountered
53 self.action = random.choice(self.actions)
54 self.display(2, f"Initial State: {state} Action {self.action}")
55 self.display(2, "s\ta\t\r\t's'\tQ")
56 return self.action
```

```
rlModelLearner.py — (continued)
```

```
58 def select_action(self, reward, next_state):
59     """do num_steps of interaction with the environment
60     for each action, do updates_per_step iterations of asynchronous
        value iteration
61     """
62     if next_state not in self.visits: # has not been encountered before
63         self.states_list.append(next_state)
64         self.visits[next_state] = {a:0 for a in self.actions}
65         self.T[next_state] = {a:{} for a in self.actions}
66         self.Q[next_state] = {a:self.Qinit for a in self.actions}
67         self.R[next_state] = {a:0 for a in self.actions}
68     if next_state in self.T[self.state][self.action]:
69         self.T[self.state][self.action][next_state] += 1
70     else:
71         self.T[self.state][self.action][next_state] = 1
72     self.visits[self.state][self.action] += 1
73     self.R[self.state][self.action] +=
        (reward-self.R[self.state][self.action])/self.visits[self.state][self.action]
74     st,act = self.state,self.action #initial state-action pair for AVI
75     for update in range(self.updates_per_step):
76         self.Q[st][act] = self.R[st][act]+self.discount*(
77             sum(self.T[st][act][nst]/self.visits[st][act]*
78                 max(self.Q[nst][nact] for nact in self.actions)
79                 for nst in self.T[st][act].keys()))
80         st = random.choice(self.states_list)
81         act = random.choice(self.actions)
82     self.state = next_state
83     self.action = self.exploration_strategy(self.Q[next_state],
84                                             self.visits[next_state],**self.es_kwargs)
85     return self.action
```

```
rlModelLearner.py — (continued)
```

```
87 from rlExamples import Monster_game_env
88 mon_env = Monster_game_env()
```

```

89 | mbl1 = Model_based_reinforcement_learner(mon_env.name, mon_env.actions,
    | 0.9, updates_per_step=10, method="model-based(10)")
90 | # Simulate(mbl1, mon_env).go(100000).plot()
91 | mbl2 = Model_based_reinforcement_learner(mon_env.name, mon_env.actions,
    | 0.9, updates_per_step=1, method="model-based(1)")
92 | # 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_____
11 | from rlExamples import Monster_game_env
12 | from rlProblem import RL_env
13 |
14 | def get_features(state, action):
15 |     """returns the list of feature values for the state-action pair
16 |     """

```



```

17     assert action in Monster_game_env.actions, f"Monster game, unknown
    action: {action}"
18     (x,y,d,p) = state
19     # f1: would go to a monster
20     f1 = monster_ahead(x,y,action)
21     # f2: would crash into wall
22     f2 = wall_ahead(x,y,action)
23     # f3: action is towards a prize
24     f3 = towards_prize(x,y,action,p)
25     # f4: damaged and action is toward repair station
26     f4 = towards_repair(x,y,action) if d else 0
27     # f5: damaged and towards monster
28     f5 = 1 if d and f1 else 0
29     # f6: damaged
30     f6 = 1 if d else 0
31     # f7: not damaged
32     f7 = 1-f6
33     # f8: damaged and prize ahead
34     f8 = 1 if d and f3 else 0
35     # f9: not damaged and prize ahead
36     f9 = 1 if not d and f3 else 0
37     features = [1,f1,f2,f3,f4,f5,f6,f7,f8,f9]
38     # the next 20 features are for 5 prize locations
39     # and 4 distances from outside in all directions
40     for pr in Monster_game_env.prize_locs+[None]:
41         if p==pr:
42             features += [x, 4-x, y, 4-y]
43         else:
44             features += [0, 0, 0, 0]
45     # fp04 feature for y when prize is at 0,4
46     # this knows about the wall to the right of the prize
47     if p==(0,4):
48         if x==0:
49             fp04 = y
50         elif y<3:
51             fp04 = y
52         else:
53             fp04 = 4-y
54     else:
55         fp04 = 0
56     features.append(fp04)
57     return features
58
59 def monster_ahead(x,y,action):
60     """returns 1 if the location expected to get to by doing
61     action from (x,y) can contain a monster.
62     """
63     if action == "right" and (x+1,y) in Monster_game_env.monster_locs:
64         return 1
65     elif action == "left" and (x-1,y) in Monster_game_env.monster_locs:

```

```

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

```

```

114         return 0
115
116     def towards_repair(x,y,action):
117         """returns 1 if action is towards the repair station.
118         """
119         if action == "up" and (x>0 and y<4 or x==0 and y<2):
120             return 1
121         elif action == "left" and x>1:
122             return 1
123         elif action == "right" and x==0 and y<3:
124             return 1
125         elif action == "down" and x==0 and y>2:
126             return 1
127         else:
128             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)_____
130 def simp_features(state,action):
131     """returns a list of feature values for the state-action pair
132     """
133     assert action in Monster_game_env.actions
134     (x,y,d,p) = state
135     # f1: would go to a monster
136     f1 = monster_ahead(x,y,action)
137     # f2: would crash into wall
138     f2 = wall_ahead(x,y,action)
139     # f3: action is towards a prize
140     f3 = towards_prize(x,y,action,p)
141     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_____
11 import random
12 from rlProblem import RL_agent, epsilon_greedy, ucb
13 from display import Displayable
14 from utilities import argmaxe, flip
15
16 class SARSA_LFA_learner(RL_agent):
17     """A SARSA with linear function approximation (LFA) learning agent has
18     """
19     def __init__(self, role, actions, discount, get_features,

```

```

20         exploration_strategy=epsilon_greedy, es_kwargs={},
21         step_size=0.01, winit=0, method="SARSA_LFA"):
22     """role is the role of the agent (e.g., in a game)
23     actions is the set of actions the agent can do
24     discount is the discount factor
25     get_features is a function get_features(state,action) -> list of
        feature values
26     exploration_strategy is the exploration function, default
        "epsilon_greedy"
27     es_kwargs is extra keyword arguments of the exploration_strategy
28     step_size is gradient descent step size
29     winit is the initial value of the weights
30     method gives the method used to implement the role (for plotting)
31     """
32     RL_agent.__init__(self, actions)
33     self.role = role
34     self.discount = discount
35     self.exploration_strategy = exploration_strategy
36     self.es_kwargs = es_kwargs
37     self.get_features = get_features
38     self.step_size = step_size
39     self.winit = winit
40     self.method = method

```

The initial action is a random action. It remembers the state, and initializes the data structures.

```

rlFeatures.py — (continued)
42     def initial_action(self, state):
43         """ Returns the initial action; selected at random
44         Initialize Data Structures
45         """
46         self.action = random.choice(self.actions)
47         self.features = self.get_features(state, self.action)
48         self.weights = [self.winit for f in self.features]
49         self.state = state
50         self.display(2, f"Initial State: {state} Action {self.action}")
51         self.display(2, "s\\ta\\tr\\ts\\tQ")
52         return self.action

```

do takes in the number of steps.

```

rlFeatures.py — (continued)
54
55     def Q(self, state, action):
56         """returns Q-value of the state and action for current weights
57         """
58         return dot_product(self.weights, self.get_features(state, action))
59
60     def select_action(self, reward, next_state):
61         """do num_steps of interaction with the environment"""
62         feature_values = self.get_features(self.state, self.action)

```

```

63     oldQ = self.Q(self.state,self.action)
64     next_action = self.exploration_strategy({a:self.Q(next_state,a)
65                                             for a in self.actions}, {})
66     nextQ = self.Q(next_state,next_action)
67     delta = reward + self.discount * nextQ - oldQ
68     for i in range(len(self.weights)):
69         self.weights[i] += self.step_size * delta * feature_values[i]
70     self.display(2,self.state, self.action, reward, next_state,
71                 self.Q(self.state,self.action), delta, sep='\t')
72     self.state = next_state
73     self.action = next_action
74     return self.action
75
76     def show_actions(self,state=None):
77         """prints the value for each action in a state.
78         This may be useful for debugging.
79         """
80         if state is None:
81             state = self.state
82         for next_act in self.actions:
83             print(next_act,dot_product(self.weights,
84                                       self.get_features(state,next_act)))
85
86     def dot_product(l1,l2):
87         return sum(e1*e2 for (e1,e2) in zip(l1,l2))

```

Test code:

```

r1Features.py — (continued)
88 from rlProblem import Simulate
89 from rlExamples import Monster_game_env # monster game environment
90 import rlMonsterGameFeatures
91
92 mon_env = Monster_game_env()
93 fa1 = SARSA_LFA_learner(mon_env.name, mon_env.actions, 0.9,
94                         rlMonsterGameFeatures.get_features)
95 # Simulate(fa1,mon_env).go(100000).plot()
96 fas1 = SARSA_LFA_learner(mon_env.name, mon_env.actions, 0.9,
97                          rlMonsterGameFeatures.simp_features, method="LFA (simp features)")
98 #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 from the current state); how could the algorithm be modified to make sure it has at least explored the close neighborhood of the current state?

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

```

29
30     def children(self):
31         """returns the list of all children."""
32         return self.allchildren
33
34     def evaluate(self):
35         """returns the evaluation for this node if it is a leaf"""
36         return self.value

```

The following gives the tree from Figure 11.5 of the book. Note how 888 is used as a value here, but never appears in the trace.

```

masProblem.py — (continued)
38 fig10_5 = Node("a", True, None, [
39     Node("b", False, None, [
40         Node("d", True, None, [
41             Node("h", False, None, [
42                 Node("h1", True, 7, None),
43                 Node("h2", True, 9, None)]],
44             Node("i", False, None, [
45                 Node("i1", True, 6, None),
46                 Node("i2", True, 888, None)]]),
47         Node("e", True, None, [
48             Node("j", False, None, [
49                 Node("j1", True, 11, None),
50                 Node("j2", True, 12, None)]],
51         Node("k", False, None, [
52             Node("k1", True, 888, None),
53             Node("k2", True, 888, None)]])]),
54     Node("c", False, None, [
55         Node("f", True, None, [
56             Node("l", False, None, [
57                 Node("l1", True, 5, None),
58                 Node("l2", True, 888, None)]],
59         Node("m", False, None, [
60             Node("m1", True, 4, None),
61             Node("m2", True, 888, None)]]),
62     Node("g", True, None, [
63         Node("n", False, None, [
64             Node("n1", True, 888, None),
65             Node("n2", True, 888, None)]],
66     Node("o", False, None, [
67         Node("o1", True, 888, None),
68         Node("o2", True, 888, None)]])])])])

```

The following is a representation of a **magic-sum game**, where players take turns picking a number in the range [1, 9], and the first player to have 3 numbers that sum to 15 wins. Note that this is a syntactic variant of **tic-tac-toe** or **naughts and crosses**. To see this, consider the numbers on a **magic square** (Figure 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)

```

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

```

```

105         x = self.x,
106         o = self.o+[sel])
107         for sel in self.available]
108     return self.allchildren
109
110     def isLeaf(self):
111         """A leaf has no numbers available or is a win for one of the
112         players.
113         We only need to check for a win for o if it is currently x's turn,
114         and only check for a win for x if it is o's turn (otherwise it would
115         have been a win earlier).
116         """
117         return (self.available == [] or
118                 (sum_to_15(self.last_move,self.o)
119                  if self.xmove
120                  else sum_to_15(self.last_move,self.x)))
121
122     def evaluate(self):
123         if self.xmove and sum_to_15(self.last_move,self.o):
124             return -1
125         elif not self.xmove and sum_to_15(self.last_move,self.x):
126             return 1
127         else:
128             return 0
129
130     def sum_to_15(last,selected):
131         """is true if last, together with two other elements of selected sum to
132         15.
133         """
134         return any(last+a+b == 15
135                    for a in selected if a != last
136                    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
11 def minimax(node,depth):
12     """returns the value of node, and a best path for the agents
13     """
14     if node.isLeaf():
15         return node.evaluate(),None
16     elif node.isMax:
17         max_score = float("-inf")
18         max_path = None
19         for C in node.children():
20             score,path = minimax(C,depth+1)
21             if score > max_score:
22                 max_score = score
23                 max_path = C.name,path

```

```

24         return max_score,max_path
25     else:
26         min_score = float("inf")
27         min_path = None
28         for C in node.children():
29             score,path = minimax(C,depth+1)
30             if score < min_score:
31                 min_score = score
32                 min_path = C.name,path
33     return min_score,min_path

```

The following is a depth-first minimax with α - β pruning. It returns the value for a node as well as a best path for the agents.

```

masMiniMax.py — (continued)
35 def minimax_alpha_beta(node,alpha,beta,depth=0):
36     """node is a Node, alpha and beta are cutoffs, depth is the depth
37     returns value, path
38     where path is a sequence of nodes that results in the value
39     """
40     node.display(2," "*depth,"minimax_alpha_beta(",node.name,", ",alpha, ",
41     ", beta,")")
42     best=None # only used if it will be pruned
43     if node.isLeaf():
44         node.display(2," "*depth,"returning leaf value",node.evaluate())
45         return node.evaluate(),None
46     elif node.isMax:
47         for C in node.children():
48             score,path = minimax_alpha_beta(C,alpha,beta,depth+1)
49             if score >= beta: # beta pruning
50                 node.display(2," "*depth,"pruned due to
51                 beta=",beta,"C=",C.name)
52                 return score, None
53             if score > alpha:
54                 alpha = score
55                 best = C.name, path
56             node.display(2," "*depth,"returning max alpha",alpha,"best",best)
57             return alpha,best
58     else:
59         for C in node.children():
60             score,path = minimax_alpha_beta(C,alpha,beta,depth+1)
61             if score <= alpha: # alpha pruning
62                 node.display(2," "*depth,"pruned due to
63                 alpha=",alpha,"C=",C.name)
64                 return score, None
65             if score < beta:
66                 beta=score
67                 best = C.name,path
68             node.display(2," "*depth,"returning min beta",beta,"best=",best)
69             return beta,best

```

Testing:

```

masMiniMax.py — (continued)
68 from masProblem import fig10_5, Magic_sum, Node
69
70 # Node.max_display_level=2 # print detailed trace
71 # minimax_alpha_beta(fig10_5, -9999, 9999,0)
72 # minimax_alpha_beta(Magic_sum(), -9999, 9999,0)
73
74 #To see how much time alpha-beta pruning can save over minimax, uncomment
  the following:
75 ## import timeit
76 ## timeit.Timer("minimax(Magic_sum(),0)",setup="from __main__ import
  minimax, Magic_sum"
77 ##                      ).timeit(number=1)
78 ## trace=False
79 ## timeit.Timer("minimax_alpha_beta(Magic_sum(), -9999, 9999,0)",
80 ##                      setup="from __main__ import minimax_alpha_beta, Magic_sum"
81 ##                      ).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
11 from display import Displayable
12 import utilities # argmaxall for (element,value) pairs
13 import matplotlib.pyplot as plt
14 import random
15 from rlProblem import RL_agent
16
17
18 class StochasticPIAgent(RL_agent):
19     """This agent maintains the Q-function for each state.
20     Chooses the best action using empirical distribution over actions
21     """
22     def __init__(self, role, actions, discount=0,
23                 alpha_fun=lambda k:10/(9+k), Qinit=1, pi_init=1,
24                 method="Stochastic Q_learner"):
25         """
26         role is the role of the agent (e.g., in a game)
27         actions is the set of actions the agent can do.
28         discount is the discount factor (0 is appropriate if there is a
29         single state)
30         alpha_fun is a function that computes alpha from the number of
31         visits

```

```

29     Qinit is the initial q-values
30     pi_init gives the prior counts (Dirichlet prior) for the policy
31         (must be >0)
32     method gives the method used to implement the role
33     """
34     #self.max_display_level = 3
35     RL_agent.__init__(self, actions)
36     self.role = role
37     self.discount = discount
38     self.alpha_fun = alpha_fun
39     self.Qinit = Qinit
40     self.pi_init = pi_init
41     self.method = method
42     self.Q = {}
43     self.pi = {}
44     self.visits = {}
45
46     def initial_action(self, state):
47         """ Returns the initial action; selected at random
48         Initialize Data Structures
49         """
50         self.Q[state] = {act:self.Qinit for act in self.actions}
51         self.pi[state] = {act:self.pi_init for act in self.actions}
52         self.action = random.choice(self.actions)
53         self.visits[state] = {act:0 for act in self.actions}
54         self.state = state
55         self.display(2, f"Initial State: {state} Action {self.action}")
56         self.display(2, "s\\ta\\tr\\ts\\tQ")
57         return self.action
58
59     def select_action(self, reward, next_state):
60         """give reward and next state, select next action to be carried
61         out"""
62         if next_state not in self.visits: # next state not seen before
63             self.Q[next_state] = {act:self.Qinit for act in self.actions}
64             self.pi[next_state] = {act:self.pi_init for act in
65                 self.actions}
66             self.visits[next_state] = {act:0 for act in self.actions}
67         self.visits[self.state][self.action] +=1
68         alpha = self.alpha_fun(self.visits[self.state][self.action])
69         self.Q[self.state][self.action] += alpha*(
70             reward
71             + self.discount * max(self.Q[next_state].values())
72             - self.Q[self.state][self.action])
73         a_best = utilities.argmaxd(self.Q[self.state])
74         self.pi[self.state][a_best] +=1
75         self.display(2,self.state, self.action, reward, next_state,
76             self.Q[self.state][self.action], sep='\\t')
77         self.state = next_state
78         self.action = select_from_dist(self.pi[next_state])

```

```

77         self.display(3,f"Agent {self.role} doing {self.action} in state
78         {self.state}")
79         return self.action
80
81     def normalize(dist):
82         """dict is a {value:number} dictionary, where the numbers are all
83         non-negative
84         returns dict where the numbers sum to one
85         """
86         tot = sum(dist.values())
87         return {var:val/tot for (var,val) in dist.items()}
88
89     def select_from_dist(dist):
90         rand = random.random()
91         for (act,prob) in normalize(dist).items():
92             rand -= prob
93             if rand < 0:
94                 return act

```

The agent can be tested on the reinforcement learning benchmarks from the previous chapter:

```

masLearn.py — (continued)
95 ##### Testing on RL benchmarks #####
96 from rlProblem import Simulate
97 from rlExamples import Healthy_env, Monster_game_env
98 mon_env = Monster_game_env()
99 magspi = StochasticPIAgent(mon_env.name, mon_env.actions,0.9)
100 #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)
102 class SimulateGame(Displayable):
103     def __init__(self, game, agent_types):
104         #self.max_display_level = 3
105         self.game = game
106         self.agents = [agent_types[i](game.players[i], game.actions[i], 0)
107             for i in range(game.num_agents)] # list of agents
108         self.action_dists = [{act:0 for act in game.actions[i]} for i in
109             range(game.num_agents)]
110         self.action_history = []
111         self.state_history = []
112         self.reward_history = []
113         self.dist = {}
114         self.dist_history = []
115         self.actions = tuple(ag.initial_action(game.initial_state) for ag
116             in self.agents)

```

```

114         self.num_steps = 0
115
116     def go(self, steps):
117         for i in range(steps):
118             self.num_steps += 1
119             (self.rewards, state) = self.game.play(self.actions)
120             self.display(3, f"In go rewards={self.rewards}, state={state}")
121             self.reward_history.append(self.rewards)
122             self.state_history.append(state)
123             self.actions = tuple(agent.select_action(reward, state)
124                                for (agent, reward) in
125                                    zip(self.agents, self.rewards))
126             self.action_history.append(self.actions)
127             for i in range(self.game.num_agents):
128                 self.action_dists[i][self.actions[i]] += 1
129                 self.dist_history.append([a:i for (a,i) in elt.items() for
130                     elt in self.action_dists]) # deep copy
131             #print("Scores:", ' '.join(f"{self.agents[i].role} average
132                 reward={ag.total_score/self.num_steps}" for ag in self.agents))
133             print("Distributions:", '
134                 '.join(str({a:self.dist_history[-1][i][a]/sum(self.dist_history[-1][i].values())
135                     for a in self.game.actions[i]})
136                        for i in
137                            range(self.game.num_agents)))
138             #return self.reward_history, self.action_history
139
140     def action_dist(self, which_actions=[1,1]):
141         """ which actions is [a0,a1]
142         returns the empirical distribution of actions for agents,
143         where ai specifies the index of the actions for agent i
144         remove this???
145         """
146         return [sum(1 for a in sim.action_history
147                    if
148                        a[i]==gm.actions[i][which_actions[i]])/len(sim.action_history)
149                 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)

```

144     def plot_dynamics(self, x_action=0, y_action=0):
145         plt.ion() # make it interactive
146         agents = self.agents
147         x_act = self.game.actions[0][x_action]
148         y_act = self.game.actions[1][y_action]
149         plt.xlabel(f"Probability {self.game.players[0]}
150             {self.agents[0].actions[x_action]}")
151         plt.ylabel(f"Probability {self.game.players[1]}
152             {self.agents[1].actions[y_action]}")

```

```

151     plt.plot([self.dist_history[i][0][x_act]/sum(self.dist_history[i][0].values())
152              for i in range(len(self.dist_history))],
153              [self.dist_history[i][1][y_act]/sum(self.dist_history[i][1].values())
154              for i in range(len(self.dist_history))])
153     plt.legend()
154     plt.savefig('soccerplot.pdf')
155     plt.show()

```

masLearn.py — (continued)

```

157
158 class ShoppingGame(Displayable):
159     def __init__(self):
160         self.num_agents = 2
161         self.states = ['s']
162         self.initial_state = 's'
163         self.actions = [['shopping', 'football']]*2
164         self.players = ['football-preferred goes to', 'shopping-preferred
165                        goes to']
166
167     def play(self, actions):
168         """Given (action1,action2) returns (resulting_state, (reward1,
169            reward2))
170         """
171         return ({('football', 'football'): (2, 1),
172                ('football', 'shopping'): (0, 0),
173                ('shopping', 'football'): (0, 0),
174                ('shopping', 'shopping'): (1, 2)
175                }[actions], 's')
176
177 class SoccerGame(Displayable):
178     def __init__(self):
179         self.num_agents = 2
180         self.states = ['s']
181         self.initial_state = 's'
182         self.initial_state = 's'
183         self.actions = [['right', 'left']]*2
184         self.players = ['goalkeeper', 'kicker']
185
186     def play(self, actions):
187         """Given (action1,action2) returns (resulting_state, (reward1,
188            reward2))
189         resulting state is 's'
190         """
191         return ({('left', 'left'): (0.6, 0.4),
192                ('left', 'right'): (0.3, 0.7),
193                ('right', 'left'): (0.2, 0.8),
194                ('right', 'right'): (0.9, 0.1)
195                }[actions], 's')

```



```

195 class GameShow(Displayable):
196     def __init__(self):
197         self.num_agents = 2
198         self.states = ['s']
199         self.initial_state = 's'
200         self.actions = [['takes', 'gives']]*2
201         self.players = ['Agent 1', 'Agent 2']
202
203     def play(self, actions):
204         return ({('takes', 'takes'): (1, 1),
205                 ('takes', 'gives'): (11, 0),
206                 ('gives', 'takes'): (0, 11),
207                 ('gives', 'gives'): (10, 10)
208                }[actions], 's')
209
210
211 class UniqueNEGameExample(Displayable):
212     def __init__(self):
213         self.num_agents = 2
214         self.states = ['s']
215         self.initial_state = 's'
216         self.actions = [['a1', 'b1', 'c1'], ['d2', 'e2', 'f2']]
217         self.players = ['agent 1 does', 'agent 2 does']
218
219     def play(self, actions):
220         return ({('a1', 'd2'): (3, 5),
221                 ('a1', 'e2'): (5, 1),
222                 ('a1', 'f2'): (1, 2),
223                 ('b1', 'd2'): (1, 1),
224                 ('b1', 'e2'): (2, 9),
225                 ('b1', 'f2'): (6, 4),
226                 ('c1', 'd2'): (2, 6),
227                 ('c1', 'e2'): (4, 7),
228                 ('c1', 'f2'): (0, 8)
229                }[actions], 's')

```

```

232 # Choose one:
233 # gm = ShoppingGame()
234 # gm = SoccerGame()
235 # gm = GameShow()
236 # gm = UniqueNEGameExample()
237
238 from rlQLearner import Q_learner
239 from rlProblem import RL_agent
240 # Choose one of the combinations of learners:
241 # sim=SimulateGame(gm,[StochasticPIAgent, StochasticPIAgent]);
242     # sim.go(10000)
243 # sim= SimulateGame(gm,[Q_learner, Q_learner]); sim.go(10000)
244 # sim=SimulateGame(gm,[Q_learner, StochasticPIAgent]); sim.go(10000)

```

```
244 |
245 |
246 | # sim.plot_dynamics()
247 |
248 | # empirical proportion that agents did their action at index 1:
249 | # sim.action_dist([1,1])
250 |
251 | # (unnormalized) empirical distribution for agent 0
252 | # 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?

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 _____
11 import random
12 import matplotlib.pyplot as plt
13 import urllib.request
14 from learnProblem import Learner
15 from display import Displayable
16
17 class CF_learner(Learner):
18     def __init__(self,
19                 rating_set,          # a Rating_set object
20                 rating_subset = None, # subset of ratings to be used as
                                     training ratings
21                 test_subset = None,  # subset of ratings to be used as test
                                     ratings
22                 step_size = 0.01,    # gradient descent step size
23                 reglz = 1.0,         # the weight for the regularization
                                     terms
24                 num_properties = 10, # number of hidden properties
25                 property_range = 0.02 # properties are initialized to be
                                     between
26                                     # -property_range and property_range
```

```

27         ):
28         self.rating_set = rating_set
29         self.ratings = rating_subset or rating_set.training_ratings #
30             whichever is not empty
31         if test_subset is None:
32             self.test_ratings = self.rating_set.test_ratings
33         else:
34             self.test_ratings = test_subset
35         self.step_size = step_size
36         self.reglz = reglz
37         self.num_properties = num_properties
38         self.num_ratings = len(self.ratings)
39         self.ave_rating = (sum(r for (u,i,r,t) in self.ratings)
40                             /self.num_ratings)
41         self.users = {u for (u,i,r,t) in self.ratings}
42         self.items = {i for (u,i,r,t) in self.ratings}
43         self.user_bias = {u:0 for u in self.users}
44         self.item_bias = {i:0 for i in self.items}
45         self.user_prop = {u:[random.uniform(-property_range,property_range)
46                               for p in range(num_properties)]
47                             for u in self.users}
48         self.item_prop = {i:[random.uniform(-property_range,property_range)
49                               for p in range(num_properties)]
50                             for i in self.items}
51         self.zeros = [0 for p in range(num_properties)]
52         self.iter=0
53     def stats(self):
54         self.display(1,"ave sumsq error of mean for training=",
55                     sum((self.ave_rating-rating)**2 for
56                         (user,item,rating,timestamp)
57                         in self.ratings)/len(self.ratings))
58         self.display(1,"ave sumsq error of mean for test=",
59                     sum((self.ave_rating-rating)**2 for
60                         (user,item,rating,timestamp)
61                         in self.test_ratings)/len(self.test_ratings))
62         self.display(1,"error on training set",
63                     self.evaluate(self.ratings))
64         self.display(1,"error on test set",
65                     self.evaluate(self.test_ratings))

```

learn carries out *num_iter* steps of gradient descent.

relnCollFilt.py — (continued)

```

65     def prediction(self,user,item):
66         """Returns prediction for this user on this item.
67         The use of .get() is to handle users or items not in the training
68         set.
69         """
70         return (self.ave_rating
71                 + self.user_bias.get(user,0) #self.user_bias[user]

```

```

71         + self.item_bias.get(item,0) #self.item_bias[item]
72         +
73         sum([self.user_prop.get(user,self.zeros)[p]*self.item_prop.get(item,self.zeros)
74             for p in range(self.num_properties)]))
75
76 def learn(self, num_iter = 50):
77     """ do num_iter iterations of gradient descent."""
78     for i in range(num_iter):
79         self.iter += 1
80         abs_error=0
81         sumsq_error=0
82         for (user,item,rating,timestamp) in
83             random.sample(self.ratings,len(self.ratings)):
84             error = self.prediction(user,item) - rating
85             abs_error += abs(error)
86             sumsq_error += error * error
87             self.user_bias[user] -= self.step_size*error
88             self.item_bias[item] -= self.step_size*error
89             for p in range(self.num_properties):
90                 self.user_prop[user][p] -=
91                     self.step_size*error*self.item_prop[item][p]
92                 self.item_prop[item][p] -=
93                     self.step_size*error*self.user_prop[user][p]
94         for user in self.users:
95             self.user_bias[user] -= self.step_size*self.reglz*
96                 self.user_bias[user]
97             for p in range(self.num_properties):
98                 self.user_prop[user][p] -=
99                     self.step_size*self.reglz*self.user_prop[user][p]
100         for item in self.items:
101             self.item_bias[item] -=
102                 self.step_size*self.reglz*self.item_bias[item]
103             for p in range(self.num_properties):
104                 self.item_prop[item][p] -=
105                     self.step_size*self.reglz*self.item_prop[item][p]
106         self.display(1,"Iteration",self.iter,
107             "(Ave Abs,AveSumSq) training",self.evaluate(self.ratings),
108             "test =",self.evaluate(self.test_ratings))

```

evaluate evaluates current predictions on the rating set:

```

relnCollFilt.py — (continued)
102 def evaluate(self,ratings):
103     """returns (average_absolute_error, average_sum_squares_error) for
104         ratings
105     """
106     abs_error = 0
107     sumsq_error = 0
108     if not ratings: return (0,0)
109     for (user,item,rating,timestamp) in ratings:

```

```

109         error = self.prediction(user,item) - rating
110         abs_error += abs(error)
111         sumsq_error += error * error
112     return abs_error/len(ratings), sumsq_error/len(ratings)

```

15.1.1 Alternative Formulation

An alternative formulation is to regularize after each update.

15.1.2 Plotting

```

relnCollFilt.py — (continued)
114 def plot_predictions(self, examples="test"):
115     """
116     examples is either "test" or "training" or the actual examples
117     """
118     if examples == "test":
119         examples = self.test_ratings
120     elif examples == "training":
121         examples = self.ratings
122     plt.ion()
123     plt.xlabel("prediction")
124     plt.ylabel("cumulative proportion")
125     self.actuals = [[] for r in range(0,6)]
126     for (user,item,rating,timestamp) in examples:
127         self.actuals[rating].append(self.prediction(user,item))
128     for rating in range(1,6):
129         self.actuals[rating].sort()
130         numrat=len(self.actuals[rating])
131         yvals = [i/numrat for i in range(numrat)]
132         plt.plot(self.actuals[rating], yvals,
133                 label="rating="+str(rating))
134     plt.legend()
135     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)
136 def plot_property(self,
137                 p, # property
138                 plot_all=False, # true if all points should be plotted
139                 num_points=200 # number of random points plotted if not
140                             all
141                 ):
142     """plot some of the user-movie ratings,

```

```

142         if plot_all is true
143             num_points is the number of points selected at random plotted.
144
145         the plot has the users on the x-axis sorted by their value on
146             property p and
147         with the items on the y-axis sorted by their value on property p and
148         the ratings plotted at the corresponding x-y position.
149         """
150         plt.ion()
151         plt.xlabel("users")
152         plt.ylabel("items")
153         user_vals = [self.user_prop[u][p]
154                     for u in self.users]
155         item_vals = [self.item_prop[i][p]
156                    for i in self.items]
157         plt.axis([min(user_vals)-0.02,
158                 max(user_vals)+0.05,
159                 min(item_vals)-0.02,
160                 max(item_vals)+0.05])
161         if plot_all:
162             for (u,i,r,t) in self.ratings:
163                 plt.text(self.user_prop[u][p],
164                         self.item_prop[i][p],
165                         str(r))
166         else:
167             for i in range(num_points):
168                 (u,i,r,t) = random.choice(self.ratings)
169                 plt.text(self.user_prop[u][p],
170                         self.item_prop[i][p],
171                         str(r))
172         plt.show()

```

15.1.3 Creating Rating Sets

A rating set can be read from the Internet or read from a local file. The default is to read the Movielens 100K dataset from the Internet. It would be more efficient to save the dataset as a local file, and then set *local_file = True*, as then it will not need to download the dataset every time the program is run.

```

_____relnCollFilt.py — (continued) _____
173 class Rating_set(Displayable):
174     def __init__(self,
175                 date_split=892000000,
176                 local_file=False,
177                 url="http://files.grouplens.org/datasets/movielens/ml-100k/u.data",
178                 file_name="u.data"):
179         self.display(1,"reading...")
180         if local_file:
181             lines = open(file_name,'r')
182         else:

```

```

183         lines = (line.decode('utf-8') for line in
184                   urllib.request.urlopen(url))
185     all_ratings = (tuple(int(e) for e in line.strip().split('\t'))
186                   for line in lines)
187     self.training_ratings = []
188     self.training_stats = {1:0, 2:0, 3:0, 4:0, 5:0}
189     self.test_ratings = []
190     self.test_stats = {1:0, 2:0, 3:0, 4:0, 5:0}
191     for rate in all_ratings:
192         if rate[3] < date_split: # rate[3] is timestamp
193             self.training_ratings.append(rate)
194             self.training_stats[rate[2]] += 1
195         else:
196             self.test_ratings.append(rate)
197             self.test_stats[rate[2]] += 1
198     self.display(1, "...read:", len(self.training_ratings), "training
199 ratings and",
200                 len(self.test_ratings), "test ratings")
201     tr_users = {user for (user, item, rating, timestamp) in
202                 self.training_ratings}
203     test_users = {user for (user, item, rating, timestamp) in
204                  self.test_ratings}
205     self.display(1, "users:", len(tr_users), "training,", len(test_users), "test,",
206                 len(tr_users & test_users), "in common")
207     tr_items = {item for (user, item, rating, timestamp) in
208                 self.training_ratings}
209     test_items = {item for (user, item, rating, timestamp) in
210                  self.test_ratings}
211     self.display(1, "items:", len(tr_items), "training,", len(test_items), "test,",
212                 len(tr_items & test_items), "in common")
213     self.display(1, "Rating statistics for training set:
214                  ", self.training_stats)
215     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)

```

210     def create_top_subset(self, num_items = 30, num_users = 30):
211         """Returns a subset of the ratings by picking the most rated items,
212         and then the users that have most ratings on these, and then all of
213         the
214         ratings that involve these users and items.
215         """
216         items = {item for (user, item, rating, timestamp) in
217                  self.training_ratings}

```



```

217     item_counts = {i:0 for i in items}
218     for (user,item,rating,timestamp) in self.training_ratings:
219         item_counts[item] += 1
220
221     items_sorted = sorted((item_counts[i],i) for i in items)
222     top_items = items_sorted[-num_items:]
223     set_top_items = set(item for (count, item) in top_items)
224
225     users = {user for (user,item,rating,timestamp) in
226              self.training_ratings}
227     user_counts = {u:0 for u in users}
228     for (user,item,rating,timestamp) in self.training_ratings:
229         if item in set_top_items:
230             user_counts[user] += 1
231
232     users_sorted = sorted((user_counts[u],u)
233                          for u in users)
234     top_users = users_sorted[-num_users:]
235     set_top_users = set(user for (count, user) in top_users)
236     used_ratings = [ (user,item,rating,timestamp)
237                    for (user,item,rating,timestamp) in
238                      self.training_ratings
239                      if user in set_top_users and item in set_top_items]
240     return used_ratings
241
242 movielens = Rating_set()
243 learner1 = CF_learner(movielens, num_properties = 1)
244 #learner1.learn(50)
245 # learner1.plot_predictions(examples = "training")
246 # learner1.plot_predictions(examples = "test")
247 #learner1.plot_property(0)
248 #movielens_subset = movielens.create_top_subset(num_items = 20, num_users
249           = 20)
250 #learner_s = CF_learner(movielens, rating_subset=movielens_subset,
251                        test_subset=[], num_properties=1)
252 #learner_s.learn(1000)
253 #learner_s.plot_property(0,plot_all=True)

```


Version History

- 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 multi-agent 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>. 166
- Chollet, F. (2021), *Deeep Learning with Python*. Manning. 169
- Dua, D. and Graff, C. (2017), UCI machine learning repository. URL <http://archive.ics.uci.edu/ml>. 131
- 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>. 170
- 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*. 166
- Lichman, M. (2013), UCI machine learning repository. URL <http://archive.ics.uci.edu/ml>. 131
- 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. (2017), *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2nd edition, URL <https://artint.info>. 191

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, 43, 44, 60, 197, 198, 276, 277, 282, 286

Index

- α - β pruning, 299
- A* search, 45
- A* Search, 48
- action, 107
- agent, 23, 273
- argmax, 19
- assignment, 56, 183
- assumable, 101
- asynchronous value iteration, 268
- augmented feature, 142
- Bayesian network, 188
- belief network, 188
- blocks world, 110
- Boolean feature, 132
- botton-up proof, 94
- branch-and-bound search, 51
- class
 - Action_instance*, 124
 - Agent*, 24
 - Arc*, 38
 - Askable*, 91
 - Assumable*, 101
 - BNfromDBN*, 229
 - BeliefNetwork*, 188
 - Boosted_dataset*, 163
 - Boosting_learner*, 164
 - Branch_and_bound*, 88
 - CF_learner*, 307
 - CPDrename*, 227
 - CSP*, 57
 - CSP_from_STRIPS*, 120
 - Clause*, 91
 - Con_solver*, 71
 - Constraint*, 56
 - DBN*, 227
 - DBNVEfilter*, 229
 - DBNvariable*, 226
 - DF_Branch_and_bound*, 51
 - DT_learner*, 149
 - Data_from_file*, 139
 - Data_from_files*, 141
 - Data_set*, 133
 - Data_set_augmented*, 142
 - DecisionNetwork*, 250
 - DecisionVariable*, 249
 - Displayable*, 18
 - Dropout_layer*, 176
 - EM_learner*, 237
 - Env_from_MDP*, 276
 - Environment*, 24

- Evaluate*, 138
- FactorDF*, 261
- FactorMax*, 260
- FactorObserved*, 203
- FactorRename*, 226
- FactorSum*, 203
- Forward_STRIPS*, 113
- FrontierPQ*, 47
- GTB_learner*, 166
- GibbsSampling*, 213
- GraphicalModel*, 188
- GridMDP*, 263, 266
- HMM*, 216
- HMMVEfilter*, 218
- HMM_Controlled*, 220
- HMM_Local*, 221
- HMMparticleFilter*, 222
- Healthyenv*, 276
- InferenceMethod*, 195, 243
- KB*, 92
- KBA*, 101
- K_fold_dataset*, 154
- K_means_learner*, 233
- Layer*, 169
- Learner*, 144
- LikelihoodWeighting*, 209
- Linear_complete_layer*, 170
- Linear_complete_layer_RMS_Prop*, 175
- Linear_complete_layer_momentum*, 174
- Linear_learner*, 157
- MDP*, 262
- Magic_sum*, 297
- Model_based_reinforcement_learner*, 286
- Monster_game_env*, 277
- NN*, 172
- Node*, 295
- POP_node*, 124
- POP_search_from_STRIPS*, 125
- ParticleFiltering*, 210
- Path*, 40
- Planning_problem*, 108
- Plot_env*, 34
- Plot_prices*, 27
- Predict*, 145
- Prob*, 187
- ProbRC*, 198
- ProbSearch*, 197
- Q_learner*, 279
- RC_DN*, 256
- RL_agent*, 274
- RL_env*, 273
- Rating_set*, 311
- ReLU_layer*, 171
- Regression_STRIPS*, 117
- RejectionSampling*, 208
- Rob_body*, 29
- Rob_env*, 29
- Rob_middle_layer*, 32
- Rob_top_layer*, 33
- Runtime_distribution*, 85
- SARSA_LFA_learner*, 291
- SLSearcher*, 78
- STRIPS_domain*, 108
- SamplingInferenceMethod*, 207
- Search_from_CSP*, 68, 70
- Search_problem*, 37
- Search_problem_from_explicit_graph*, 39
- Search_with_AC_from_CSP*, 76
- Searcher*, 45
- SearcherMPP*, 50
- Show_Localization*, 221
- Sigmoid_layer*, 172
- SoftConstraint*, 87
- State*, 113
- Strips*, 107
- Subgoal*, 117
- TP_agent*, 27
- TP_env*, 26
- TabFactor*, 187
- Updatable_priority_queue*, 83
- Utility*, 249
- UtilityTable*, 249
- VE*, 202
- VE_DN*, 260
- Variable*, 55
- clause*, 91

- collaborative filtering, 307
- comprehensions, 12
- condition, 56
- conditional probability distribution (CPD), 185
- consistency algorithms, 71
- constraint, 56
- constraint satisfaction problem, 55
- copy_with_assign, 75
- CPD (conditional probability distribution), 185
- cross validation, 153
- CSP, 55
 - consistency, 71
 - domain splitting, 74, 76
 - search, 69
 - stochastic local search, 77
- currying, 59
- dataset, 132
- DBN
 - filtering, 229
 - unrolling, 229
- DBN (dynamic belief network), 225
- debugging, 97
- decision network, 249
- decision tree learning, 149
- decision variable, 249
- deep learning, 169
- dict_union, 20
- display, 18
- Displayable, 18
- domain splitting, 74, 76
- Dropout, 176
- dynamic belief network (DBN), 225
 - representation, 225
- EM, 237
- environment, 23, 24, 273
- error, 137
- example, 132
- explanation, 97
- explicit graph, 38
- factor, 183, 187
- factor_times, 204
- feature, 132, 134
- feature engineering, 131
- file
 - agentBuying.py*, 26
 - agentEnv.py*, 29
 - agentMiddle.py*, 32
 - agentTop.py*, 33
 - agents.py*, 24
 - cspConsistency.py*, 71
 - cspDFS.py*, 68
 - cspExamples.py*, 59
 - cspProblem.py*, 56
 - cspSLS.py*, 78
 - cspSearch.py*, 70
 - cspSoft.py*, 87
 - decnNetworks.py*, 249
 - display.py*, 18
 - learnBoosting.py*, 163
 - learnCrossValidation.py*, 154
 - learnDT.py*, 149
 - learnEM.py*, 237
 - learnKMeans.py*, 233
 - learnLinear.py*, 157
 - learnNN.py*, 169
 - learnNoInputs.py*, 145
 - learnProblem.py*, 132
 - logicAssumables.py*, 101
 - logicBottomUp.py*, 94
 - logicExplain.py*, 97
 - logicNegation.py*, 104
 - logicProblem.py*, 91
 - logicTopDown.py*, 96
 - masLearn.py*, 300
 - masMiniMax.py*, 298
 - masProblem.py*, 295
 - mdpExamples.py*, 262
 - mdpProblem.py*, 262
 - probCounterfactual.py*, 245
 - probDBN.py*, 226
 - probDo.py*, 243
 - probFactors.py*, 183
 - probGraphicalModels.py*, 188
 - probHMM.py*, 216
 - probLocalization.py*, 220
 - probRC.py*, 197

- probStochSim.py*, 206
- probVE.py*, 202
- pythonDemo.py*, 13
- relnCollFilt.py*, 307
- rlExamples.py*, 276
- rlFeatures.py*, 291
- rlModelLearner.py*, 286
- rlMonsterGameFeatures.py*, 288
- rlProblem.py*, 273
- rlQExperienceReplay.py*, 283
- rlQLearner.py*, 279
- searchBranchAndBound.py*, 51
- searchGeneric.py*, 45
- searchMPP.py*, 50
- searchProblem.py*, 37
- searchTest.py*, 53
- stripsCSPPlanner.py*, 120
- stripsForwardPlanner.py*, 113
- stripsHeuristic.py*, 115
- stripsPOP.py*, 124
- stripsProblem.py*, 107
- stripsRegressionPlanner.py*, 117
- utilities.py*, 19
- variable.py*, 55
- filtering, 218, 222
 - DBN, 229
- flip, 20
- forward planning, 112
- frange, 134
- ftype, 134
- game, 295
- Gibbs sampling, 213
- graphical model, 188
- heuristic planning, 115, 119
- hidden Markov model, 216
- hierarchical controller, 28
- HMM
 - exact filtering, 218
 - particle filtering, 222
- HMM (hidden Markov models), 216
- importance sampling, 210
- interact
 - proofs, 98
- ipython, 10
- k-means, 233
- kernel, 142
- knowledge base, 92
- learner, 144
- learning, 131–181, 233–241, 273–294, 307–313
 - cross validation, 153
 - decision tree, 149
 - deep, 169–181
 - deep learning, 169
 - EM, 237
 - k-means, 233
 - linear regression, 157
 - linear classification, 157
 - neural network, 169
 - no inputs, 145
 - reinforcement, 273–294
 - relational, 307
 - supervised, 131–168
 - with uncertainty, 233–241
- LightGBM, 166
- likelihood weighting, 209
- linear regression, 157
- linear classification, 157
- localization, 220
- logistic regression, 185
- logit, 158, 159
- loss, 137
- magic square, 296
- magic-sum game, 296
- Markov Chain Monte Carlo, 213
- Markov decision process, 262
- `max_display_level`, 18
- MCMC, 213
- MDP, 262, 276
- method
 - consistent*, 58
 - holds*, 57
 - maxh*, 115
 - zero*, 113
- minimax, 295
- minimax algorithm, 298

- minsets, 102
- model-based reinforcement learner, 285
- multiagent system, 295
- multiple path pruning, 49
- n-queens problem, 67
- naive search probabilistic inference, 197
- naughts and crosses, 296
- neural network, 169
- noisy-or, 186
- NotImplementedError, 24
- partial-order planner, 123
- particle filtering, 210
 - HMMs, 222
- planning, 107–129, 249–271
 - CSP, 120
 - decision network, 249
 - forward, 112
 - MDP, 262
 - partial order, 123
 - regression, 117
 - with certainty, 107–129
 - with learning, 285
 - with uncertainty, 249–271
- plotting
 - agents in time, 27
 - reinforcement learning, 275
 - robot environment, 34
 - run-time distribution, 85
 - stochastic simulation, 214
- predictor, 137
- Prob, 187
- probabilistic inference methods, 195
- probability, 183
- proof
 - bottom-up, 94
 - explanation, 97
 - top-down, 96
- proposition, 91
- Python, 9
- Q learning, 279
- query, 195
- queryD0, 243
- RC, 198, 256
- recursive conditioning, 198
- recursive conditioning (RC), 198
- recursive conditioning for decision networks, 256
- regression planning, 117
- reinforcement learning, 273–294
 - environment, 273
 - feature-based, 288
 - model-based, 285
 - Q-learning, 279
- rejection sampling, 208
- relational learning, 307
- ReLU, 171
- resampling, 211
- robot
 - body, 29
 - environment, 29
 - middle layer, 32
 - plotting, 34
 - top layer, 33
- robot delivery domain, 108
- run time, 16
- runtime distribution, 85
- sampling, 206
 - importance sampling, 210
 - belief networks, 207
 - likelihood weighting, 209
 - particle filtering, 210
 - rejection, 208
- scope, 56
- search, 37
 - A*, 45
 - branch-and-bound, 51
 - multiple path pruning, 49
- search_with_any_conflict, 80
- search_with_var_pq, 81
- show, 58, 189
- sigmoid, 158
- softmax, 159
- stochastic local search, 77
 - any-conflict, 80

- two-stage choice, 81
- stochastic simulation, 206
- tabular factor, 187
- test
 - SLS, 86
- tic-tac-toe, 296
- top-down proof, 96
- uncertainty, 183
- unit test, 21, 48, 67, 95, 96, 98
- unrolling
 - DBN, 229
- updatable priority queue, 83
- utility, 249
- utility table, 249
- value iteration, 265
- variable, 55
- variable elimination (VE), 202
- variable elimination for decision net-
works, 260
- VE, 202
- visualize, 18
- XGBoost, 166
- yield, 15