

Python code for Artificial Intelligence

Foundations of Computational Agents

David L. Poole and Alan K. Mackworth

Version 0.9.14 of December 9, 2024.

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

©David L Poole and Alan K Mackworth 2017-2024.

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

This document and all the code can be downloaded from
<https://artint.info/AIPython/> or from <https://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	11
1.5.1 f-strings	11
1.5.2 Lists, Tuples, Sets, Dictionaries and Comprehensions	12
1.5.3 Generators	13
1.5.4 Functions as first-class objects	14
1.6 Useful Libraries	16
1.6.1 Timing Code	16
1.6.2 Plotting: Matplotlib	16
1.7 Utilities	18
1.7.1 Display	18
1.7.2 Argmax	19
1.7.3 Probability	20
1.8 Testing Code	21
2 Agent Architectures and Hierarchical Control	25
2.1 Representing Agents and Environments	25
2.2 Paper buying agent and environment	27
2.2.1 The Environment	27
2.2.2 The Agent	28

2.2.3	Plotting	29
2.3	Hierarchical Controller	31
2.3.1	World	31
2.3.2	Body	31
2.3.3	Middle Layer	34
2.3.4	Top Layer	35
2.3.5	Plotting	36
3	Searching for Solutions	41
3.1	Representing Search Problems	41
3.1.1	Explicit Representation of Search Graph	43
3.1.2	Paths	45
3.1.3	Example Search Problems	47
3.2	Generic Searcher and Variants	54
3.2.1	Searcher	54
3.2.2	GUI for Tracing Search	55
3.2.3	Frontier as a Priority Queue	59
3.2.4	A^* Search	61
3.2.5	Multiple Path Pruning	63
3.3	Branch-and-bound Search	65
4	Reasoning with Constraints	71
4.1	Constraint Satisfaction Problems	71
4.1.1	Variables	71
4.1.2	Constraints	72
4.1.3	CSPs	73
4.1.4	Examples	76
4.2	A Simple Depth-first Solver	85
4.3	Converting CSPs to Search Problems	87
4.4	Consistency Algorithms	89
4.4.1	Direct Implementation of Domain Splitting	91
4.4.2	Consistency GUI	93
4.4.3	Domain Splitting as an interface to graph searching	96
4.5	Solving CSPs using Stochastic Local Search	97
4.5.1	Any-conflict	100
4.5.2	Two-Stage Choice	101
4.5.3	Updatable Priority Queues	103
4.5.4	Plotting Run-Time Distributions	105
4.5.5	Testing	106
4.6	Discrete Optimization	107
4.6.1	Branch-and-bound Search	108
5	Propositions and Inference	111
5.1	Representing Knowledge Bases	111
5.2	Bottom-up Proofs (with askables)	114

5.3	Top-down Proofs (with askables)	116
5.4	Debugging and Explanation	117
5.5	Assumables	121
5.6	Negation-as-failure	124
6	Deterministic Planning	127
6.1	Representing Actions and Planning Problems	127
6.1.1	Robot Delivery Domain	128
6.1.2	Blocks World	130
6.2	Forward Planning	132
6.2.1	Defining Heuristics for a Planner	135
6.3	Regression Planning	137
6.3.1	Defining Heuristics for a Regression Planner	139
6.4	Planning as a CSP	140
6.5	Partial-Order Planning	144
7	Supervised Machine Learning	151
7.1	Representations of Data and Predictions	152
7.1.1	Creating Boolean Conditions from Features	155
7.1.2	Evaluating Predictions	157
7.1.3	Creating Test and Training Sets	159
7.1.4	Importing Data From File	159
7.1.5	Augmented Features	163
7.2	Generic Learner Interface	165
7.3	Learning With No Input Features	165
7.3.1	Evaluation	168
7.4	Decision Tree Learning	169
7.5	Cross Validation and Parameter Tuning	174
7.6	Linear Regression and Classification	176
7.7	Boosting	184
7.7.1	Gradient Tree Boosting	187
8	Neural Networks and Deep Learning	189
8.1	Layers	189
8.1.1	Linear Layer	190
8.1.2	ReLU Layer	192
8.1.3	Sigmoid Layer	192
8.2	Feedforward Networks	193
8.3	Improved Optimization	195
8.3.1	Momentum	195
8.3.2	RMS-Prop	195
8.4	Dropout	196
8.5	Examples	197
9	Reasoning with Uncertainty	203

9.1	Representing Probabilistic Models	203
9.2	Representing Factors	203
9.3	Conditional Probability Distributions	205
9.3.1	Logistic Regression	206
9.3.2	Noisy-or	206
9.3.3	Tabular Factors and Prob	207
9.3.4	Decision Tree Representations of Factors	208
9.4	Graphical Models	210
9.4.1	Showing Belief Networks	212
9.4.2	Example Belief Networks	212
9.5	Inference Methods	218
9.5.1	Showing Posterior Distributions	219
9.6	Naive Search	220
9.7	Recursive Conditioning	222
9.8	Variable Elimination	226
9.9	Stochastic Simulation	230
9.9.1	Sampling from a discrete distribution	230
9.9.2	Sampling Methods for Belief Network Inference	231
9.9.3	Rejection Sampling	232
9.9.4	Likelihood Weighting	233
9.9.5	Particle Filtering	234
9.9.6	Examples	235
9.9.7	Gibbs Sampling	236
9.9.8	Plotting Behavior of Stochastic Simulators	238
9.10	Hidden Markov Models	241
9.10.1	Exact Filtering for HMMs	242
9.10.2	Localization	244
9.10.3	Particle Filtering for HMMs	247
9.10.4	Generating Examples	249
9.11	Dynamic Belief Networks	250
9.11.1	Representing Dynamic Belief Networks	250
9.11.2	Unrolling DBNs	255
9.11.3	DBN Filtering	256
10	Learning with Uncertainty	259
10.1	Bayesian Learning	259
10.2	K-means	263
10.3	EM	268
11	Causality	273
11.1	Do Questions	273
11.2	Counterfactual Reasoning	276
11.2.1	Choosing Deterministic System	276
11.2.2	Firing Squad Example	280

12 Planning with Uncertainty	283
12.1 Decision Networks	283
12.1.1 Example Decision Networks	285
12.1.2 Decision Functions	291
12.1.3 Recursive Conditioning for Decision Networks	292
12.1.4 Variable elimination for decision networks	295
12.2 Markov Decision Processes	298
12.2.1 Problem Domains	299
12.2.2 Value Iteration	308
12.2.3 Value Iteration GUI for Grid Domains	309
12.2.4 Asynchronous Value Iteration	315
13 Reinforcement Learning	317
13.1 Representing Agents and Environments	317
13.1.1 Environments	317
13.1.2 Agents	318
13.1.3 Simulating an Environment-Agent Interaction	319
13.1.4 Party Environment	321
13.1.5 Environment from a Problem Domain	322
13.1.6 Monster Game Environment	323
13.2 Q Learning	326
13.2.1 Exploration Strategies	329
13.2.2 Testing Q-learning	329
13.3 Q-learning with Experience Replay	331
13.4 Stochastic Policy Learning Agent	334
13.5 Model-based Reinforcement Learner	336
13.6 Reinforcement Learning with Features	339
13.6.1 Representing Features	339
13.6.2 Feature-based RL learner	342
13.7 GUI for RL	345
14 Multiagent Systems	353
14.1 Minimax	353
14.1.1 Creating a two-player game	354
14.1.2 Minimax and α - β Pruning	357
14.2 Multiagent Learning	359
14.2.1 Simulating Multiagent Interaction with an Environment	359
14.2.2 Example Games	362
14.2.3 Testing Games and Environments	364
15 Individuals and Relations	367
15.1 Representing Datalog and Logic Programs	367
15.2 Unification	369
15.3 Knowledge Bases	370
15.4 Top-down Proof Procedure	372

15.5	Logic Program Example	374
16	Knowledge Graphs and Ontologies	377
16.1	Triple Store	377
16.2	Integrating Datalog and Triple Store	380
17	Relational Learning	383
17.1	Collaborative Filtering	383
17.1.1	Plotting	387
17.1.2	Loading Rating Sets from Files and Websites	390
17.1.3	Ratings of top items and users	391
17.2	Relational Probabilistic Models	393
18	Version History	399
	Bibliography	401
	Index	403

Python for Artificial Intelligence

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

- Readability is more important than efficiency, although the asymptotic complexity is not compromised. AIPython is not a replacement for well-designed libraries, or optimized tools. Think of it like a model of an engine made of glass, so you can see the inner workings; don't expect it to power a big truck, but it lets you see how an engine works to power a truck.
- It uses as few libraries as possible. A reader only needs to understand Python. Libraries hide details that we make explicit. The only library used is matplotlib for plotting and drawing.

1.1 Why Python?

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

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

1.2 Getting Python

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

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

```
pip install matplotlib
```

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

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

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

```
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
```

```
16 paths have been expanded and 5 paths remain in the frontier
```

```
o103 --> o109 --> o119 --> o123 --> r123
```

```
>>> searcher2.search() # find next path
```

```

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

```

You can then interact at the last prompt.

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

The rest of this chapter is about what is special about the code for AI tools. We 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

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

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

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

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

1.5.2 Lists, Tuples, Sets, Dictionaries and Comprehensions

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

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

A list comprehension is of the form

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

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

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

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

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

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

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

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

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

1.5.3 Generators

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.5.4 Functions as first-class objects

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

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

```

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

```

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

```

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

```

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

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

1.6 Useful Libraries

1.6.1 Timing Code

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

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

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

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

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

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

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

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

1.6.2 Plotting: Matplotlib

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

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

```
pythonDemo.py — (continued)
62 | import matplotlib.pyplot as plt
63 |
```

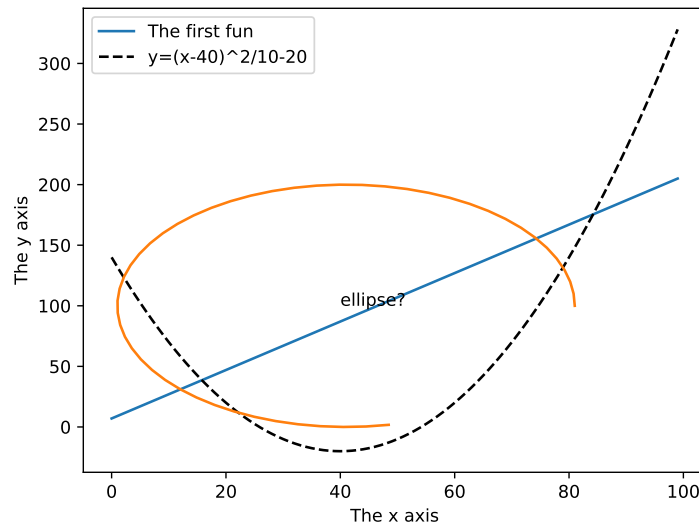



Figure 1.1: Result of pythonDemo code

```

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

```

```

91 | # plt.text(40,100,"ellipse?")
92 | # 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

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

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

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

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

The definition of `display` is:

```

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

```

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

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

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

```
Classname.max_display_level = 3
```

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

The value of `max_display_level` by convention is:

- 0 display nothing
- 1 display solutions (nothing that happens repeatedly)
- 2 also display the values as they change (little detail through a loop)
- 3 also display more details
- 4 and above even more detail

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

1.7.2 Argmax

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

```

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))

```

```

33
34 def argmax(lst):
35     """returns maximum index in a list"""
36     return argmaxe(enumerate(lst))
37 # Try:
38 # argmax([1,6,3,77,3,55,23])
39
40 def argmaxd(dct):
41     """returns the arg max of a dictionary dct"""
42     return argmaxe(dct.items())
43 # Try:
44 # arxmaxd({2:5,5:9,7:7})

```

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

1.7.3 Probability

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

```

_____utilities.py — (continued)_____
45 def flip(prob):
46     """return true with probability prob"""
47     return random.random() < prob

```

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

```

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

```

1.8 Testing Code

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

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

```

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

```

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

```

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

```

```

95     import logicBottomUp, logicTopDown, logicExplain, logicAssumables,
        logicNegation
96     logicBottomUp.test()
97     logicTopDown.test()
98     logicExplain.test()
99     logicNegation.test()
100    # Planning
101    print("\n***** testing Planning *****")
102    import stripsHeuristic
103    stripsHeuristic.test_forward_heuristic()
104    stripsHeuristic.test_regression_heuristic()
105    import stripsCSPPanner, stripsPOP
106    # Learning
107    print("\n***** testing Learning *****")
108    import learnProblem, learnNoInputs, learnDT, learnLinear
109    learnNoInputs.test_no_inputs(training_sizes=[4])
110    data = learnProblem.Data_from_file('data/carbool.csv', target_index=-1,
        seed=123)
111    learnDT.testDT(data, print_tree=False)
112    learnLinear.test()
113    import learnCrossValidation, learnBoosting
114    # Deep Learning: currently no tests
115    import learnNN
116    # Uncertainty
117    print("\n***** testing Uncertainty *****")
118    import probGraphicalModels, probRC, probVE, probStochSim
119    probGraphicalModels.InferenceMethod.testIM(probRC.ProbSearch)
120    probGraphicalModels.InferenceMethod.testIM(probRC.ProbRC)
121    probGraphicalModels.InferenceMethod.testIM(probVE.VE)
122    probGraphicalModels.InferenceMethod.testIM(probStochSim.RejectionSampling,
        threshold=0.1)
123    probGraphicalModels.InferenceMethod.testIM(probStochSim.LikelihoodWeighting,
        threshold=0.1)
124    probGraphicalModels.InferenceMethod.testIM(probStochSim.ParticleFiltering,
        threshold=0.1)
125    probGraphicalModels.InferenceMethod.testIM(probStochSim.GibbsSampling,
        threshold=0.1)
126    import probHMM, probLocalization, probDBN
127    # Learning under uncertainty: currently no tests
128    import learnBayesian, learnKMeans, learnEM
129    # Causality: currently no tests
130    import probDo, probCounterfactual
131    # Planning under uncertainty
132    print("\n***** testing Planning under Uncertainty *****")
133    import decnNetworks
134    decnNetworks.test(decnNetworks.fire_dn)
135    import mdpExamples
136    mdpExamples.test_MDP(mdpExamples.partyMDP)
137    import mdpGUI
138    # Reinforcement Learning:

```

```
139     print("\n***** testing Reinforcement Learning *****")
140     import rlQLearner
141     rlQLearner.test_RL(rlQLearner.Q_learner, alpha_fun=lambda k:10/(9+k))
142     import rlQExperienceReplay
143     rlQLearner.test_RL(rlQExperienceReplay.Q_ER_learner, alpha_fun=lambda
        k:10/(9+k))
144     import rlStochasticPolicy
145     rlQLearner.test_RL(rlStochasticPolicy.StochasticPIAgent,
        alpha_fun=lambda k:10/(9+k))
146     import rlModelLearner
147     rlQLearner.test_RL(rlModelLearner.Model_based_reinforcement_learner)
148     import rlFeatures
149     rlQLearner.test_RL(rlFeatures.SARSA_LFA_learner,
        es_kwargs={'epsilon':1}, eps=4)
150     import rlQExperienceReplay, rlModelLearner, rlFeatures, rlGUI
151     # Multiagent systems: currently no tests
152     import rlStochasticPolicy, rlGameFeature
153     # Individuals and Relations
154     print("\n***** testing Datalog and Logic Programming *****")
155     import reInExamples
156     reInExamples.test_ask_all()
157     # Knowledge Graphs and Onologies
158     print("\n***** testing Knowledge Graphs and Onologies *****")
159     import knowledgeGraph, knowledgeReasoning
160     knowledgeGraph.test_kg()
161     # Relational Learning: currently no tests
162     import reInCollFilt, reInProbModels
163     print("\n***** End of Testing*****")
```


Agent Architectures and Hierarchical Control

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

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

2.1 Representing Agents and Environments

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

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

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

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

- An agent implements the `initial_action(percept)` method which is used initially. This is the method used in the reinforcement learning chapter (page 317).
- The environment implements the `initial_percept()` method which gives the initial percept for the agent. This is the method is used in this chapter.

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

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

```

agents.py — Agent and Controllers
11 from display import Displayable
12
13 class Agent(Displayable):
14
15     def initial_action(self, percept):
16         """return the initial action."""
17         return self.select_action(percept) # same as select_action
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

```

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

```

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

```

2.2 Paper buying agent and environment

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

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

2.2.1 The Environment

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

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

```

agentBuying.py — Paper-buying agent
11 import random
12 from agents import Agent, Environment, Simulate
13 from utilities import select_from_dist

```

```

14
15 class TP_env(Environment):
16     price_delta = [0, 0, 0, 21, 0, 20, 0, -64, 0, 0, 23, 0, 0, 0, -35,
17                   0, 76, 0, -41, 0, 0, 21, 0, 5, 0, 5, 0, 0, 0, 5, 0, -15, 0, 5,
18                   0, 5, 0, -115, 0, 115, 0, 5, 0, -15, 0, 5, 0, 5, 0, 0, 0, 5, 0,
19                   -59, 0, 44, 0, 5, 0, 5, 0, 0, 0, 5, 0, -65, 50, 0, 5, 0, 5, 0, 0,
20                   0, 5, 0]
21     sd = 5 # noise standard deviation
22
23     def __init__(self):
24         """paper buying agent"""
25         self.time=0
26         self.stock=20
27         self.stock_history = [] # memory of the stock history
28         self.price_history = [] # memory of the price history
29
30     def initial_percept(self):
31         """return initial percept"""
32         self.stock_history.append(self.stock)
33         self.price = round(234+self.sd*random.gauss(0,1))
34         self.price_history.append(self.price)
35         return {'price': self.price,
36               'instock': self.stock}
37
38     def do(self, action):
39         """does action (buy) and returns percept consisting of price and
40           instock"""
41         used = select_from_dist({6:0.1, 5:0.1, 4:0.1, 3:0.3, 2:0.2, 1:0.2})
42         # used = select_from_dist({7:0.1, 6:0.2, 5:0.2, 4:0.3, 3:0.1,
43           2:0.1}) # uses more paper
44         bought = action['buy']
45         self.stock = self.stock+bought-used
46         self.stock_history.append(self.stock)
47         self.time += 1
48         self.price = round(self.price
49                           + self.price_delta[self.time%len(self.price_delta)] #
50                           repeating pattern
51                           + self.sd*random.gauss(0,1)) # plus randomness
52         self.price_history.append(self.price)
53         return {'price': self.price,
54               '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)

```

53 class TP_agent(Agent):
54     def __init__(self):
55         self.spent = 0
56         percept = env.initial_percept()
57         self.ave = self.last_price = percept['price']
58         self.instock = percept['instock']
59         self.buy_history = []
60
61     def select_action(self, percept):
62         """return next action to carry out
63         """
64         self.last_price = percept['price']
65         self.ave = self.ave+(self.last_price-self.ave)*0.05
66         self.instock = percept['instock']
67         if self.last_price < 0.9*self.ave and self.instock < 60:
68             tobuy = 48
69         elif self.instock < 12:
70             tobuy = 12
71         else:
72             tobuy = 0
73         self.spent += tobuy*self.last_price
74         self.buy_history.append(tobuy)
75         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)
77 env = TP_env()
78 ag = TP_agent()
79 sim = Simulate(ag,env)
80 #sim.go(90)
81 #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)
83 import matplotlib.pyplot as plt
84
85 class Plot_history(object):
86     """Set up the plot for history of price and number in stock"""
87     def __init__(self, ag, env):
88         self.ag = ag
89         self.env = env
90         plt.ion()
91         plt.xlabel("Time")
92         plt.ylabel("Value")
93

```

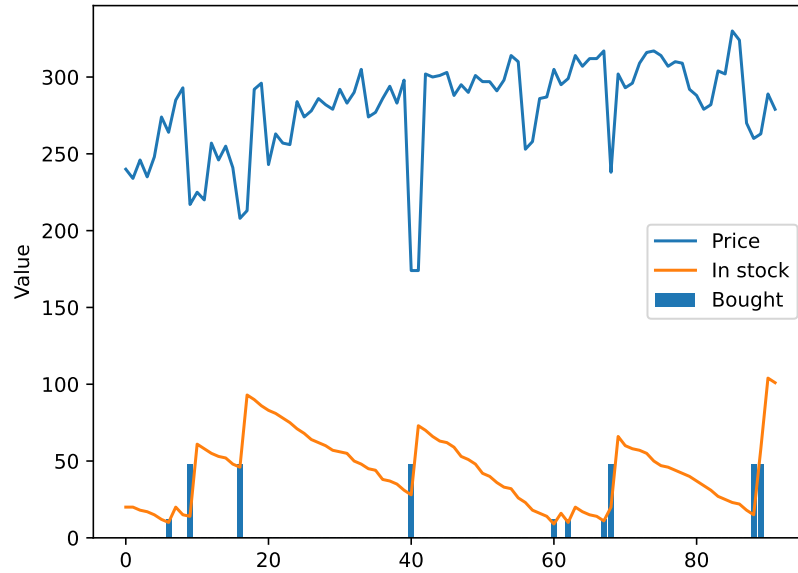


Figure 2.1: Percept and command traces for the paper-buying agent

```

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

```

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

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

- Justify a performance measure that is a fair comparison. Note that minimizing the total amount of money spent may be unfair to agents who have built up a stockpile, and favors agents that end up with no paper.
- Give a controller that can work for many different price histories. An agent

can use other local state variables, but does not have access to the environment model.

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

2.3 Hierarchical Controller

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

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

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

2.3.1 World

The world defines the walls. This is not implemented as an environment as it does not change. If the agent could move walls, it should be made into an environment.

```

agentEnv.py — Agent environment
11 import math
12 from display import Displayable
13
14 class Rob_world(Displayable):
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

`Rob_body` defines everything about the agent body, its position and orientation and whether its whisker sensor is on. It implements the Environment class as

it is treated as an environment by the higher layers. It can be told to turn left or right or to go straight.

```

agentEnv.py — (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, world, init_pos=(0,0,90)):
28         """ world is the current world
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.world = world
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
52     initial_percept = percept # use percept function for initial percept too
53
54     def do(self,action):
55         """ action is {'steer':direction}
56         direction is 'left', 'right' or 'straight'.
57         Returns current percept.
58         """
59         if self.crashed:
60             return self.percept()
61         direction = action['steer']
62         compass_deriv =
63             {'left':1,'straight':0,'right':-1}[direction]*self.turning_angle
64         self.rob_dir = (self.rob_dir + compass_deriv +360)%360 # make in
65             range [0,360)
66         rob_x_new = self.rob_x + math.cos(self.rob_dir*math.pi/180)
67         rob_y_new = self.rob_y + math.sin(self.rob_dir*math.pi/180)
68         path = ((self.rob_x,self.rob_y),(rob_x_new,rob_y_new))

```



```

63         if any(line_segments_intersect(path,wall) for wall in
64             self.world.walls):
65             self.crashed = True
66             self.display(1, "*Crashed*")
67             if self.plotting:
68                 plt.plot([self.rob_x],[self.rob_y],"r*",markersize=20.0)
69                 plt.draw()
70             self.rob_x, self.rob_y = rob_x_new, rob_y_new
71             self.history.append((self.rob_x, self.rob_y))
72             if self.plotting and not self.crashed:
73                 plt.plot([self.rob_x],[self.rob_y],"go")
74                 plt.draw()
75                 plt.pause(self.sleep_time)
76         return self.percept()

```

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

agentEnv.py — (continued)

```

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

```

```

109     ca = (db*(y0b-y0a)-eb*(x0b-x0a))/denom # intersect along line a
110     return 0<=ca<=1 # intersect is inside both line segments
111
112 # Test cases:
113 # assert line_segments_intersect(((0,0),(1,1)),((1,0),(0,1)))
114 # assert not line_segments_intersect(((0,0),(1,1)),((1,0),(0.6,0.4)))
115 # 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 body layer) and an environment for the upper layer. It has to tell the body how to steer. Thus it calls *env.do(·)*, where *env* is the body. It implements *do(·)* for the top layer, where the action specifies an *x-y* position to go to and a timeout.

```

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

```

```

42         remaining -= 1
43         arrived = self.close_enough(target_pos)
44         return {'arrived':arrived}

```

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

```

agentMiddle.py — (continued)
46 def steer(self, target_pos):
47     if self.percept['whisker']:
48         self.display(3,'whisker on', self.percept)
49         return "left"
50     else:
51         return self.head_towards(target_pos)
52
53 def head_towards(self, target_pos):
54     """ given a target position, return the action that heads
55         towards that position
56     """
57     gx,gy = target_pos
58     rx,ry = self.percept['rob_x_pos'],self.percept['rob_y_pos']
59     goal_dir = math.acos((gx-rx)/math.sqrt((gx-rx)*(gx-rx)
60                                     +(gy-ry)*(gy-ry)))*180/math.pi
61     if ry>gy:
62         goal_dir = -goal_dir
63     goal_from_rob = (goal_dir - self.percept['rob_dir']+540)%360-180
64     assert -180 < goal_from_rob <= 180
65     if goal_from_rob > self.straight_angle:
66         return "left"
67     elif goal_from_rob < -self.straight_angle:
68         return "right"
69     else:
70         return "straight"
71
72 def close_enough(self, target_pos):
73     """True when the robot's position is within close_threshold of
74         target_pos
75     """
76     gx,gy = target_pos
77     rx,ry = self.percept['rob_x_pos'],self.percept['rob_y_pos']
78     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 display import Displayable
12 from agentMiddle import Rob_middle_layer

```

```

13 from agents import Environment
14
15 class Rob_top_layer(Environment):
16     def __init__(self, middle, timeout=200, locations = {'mail':(-5,10),
17                                                         'o103':(50,10), 'o109':(100,10),'storage':(101,51)}
18                 ):
19         """middle is the middle layer
20         timeout is the number of steps the middle layer goes before giving
21         up
22         locations is a loc:pos dictionary
23         where loc is a named location, and pos is an (x,y) position.
24         """
25         self.middle = middle
26         self.timeout = timeout # number of steps before the middle layer
27                                # should give up
28         self.locations = locations
29
30     def do(self,plan):
31         """carry out actions.
32         actions is of the form {'visit':list_of_locations}
33         It visits the locations in turn.
34         """
35         to_do = plan['visit']
36         for loc in to_do:
37             position = self.locations[loc]
38             arrived = self.middle.do({'go_to':position,
39                                     'timeout':self.timeout})
40             self.display(1,"Goal",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)
38 import matplotlib.pyplot as plt
39
40 class Plot_env(Displayable):
41     def __init__(self, body,top):
42         """sets up the plot
43         """
44         self.body = body
45         self.top = top
46         plt.ion()
47         plt.axes().set_aspect('equal')
48         self.redraw()
49

```

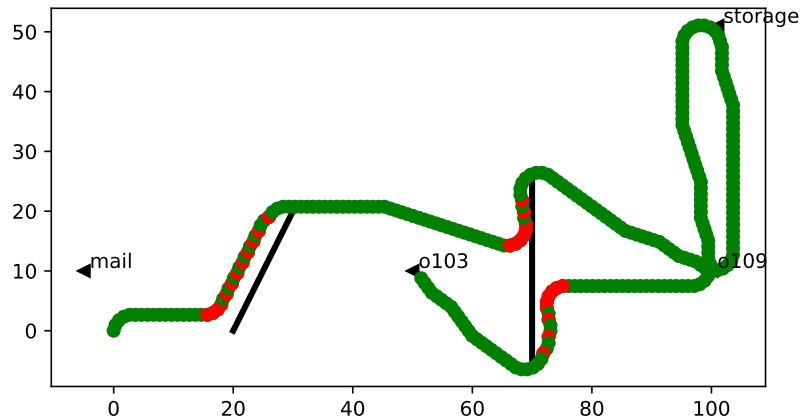


Figure 2.2: A trace of the trajectory of the agent. Red dots correspond to the whisker sensor being on; the green dot to the whisker sensor being off. The agent starts at position $(0,0)$ facing up.

```

50 def redraw(self):
51     plt.clf()
52     for wall in self.body.world.walls:
53         ((x0,y0),(x1,y1)) = wall
54         plt.plot([x0,x1],[y0,y1],"-k",linewidth=3)
55     for loc in self.top.locations:
56         (x,y) = self.top.locations[loc]
57         plt.plot([x],[y],"k<")
58         plt.text(x+1.0,y+0.5,loc) # print the label above and to the
                                     right
59     plt.plot([self.body.rob_x],[self.body.rob_y],"go")
60     plt.gca().figure.canvas.draw()
61     if self.body.history or self.body.wall_history:
62         self.plot_run()
63
64 def plot_run(self):
65     """plots the history after the agent has finished.
66     This is typically only used if body.plotting==False
67     """
68     if self.body.history:
69         xs,ys = zip(*self.body.history)
70         plt.plot(xs,ys,"go")
71     if self.body.wall_history:
72         wxs,wys = zip(*self.body.wall_history)
73         plt.plot(wxs,wys,"ro")

```

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

```
agentTop.py — (continued)
```

```
75 from agentEnv import Rob_body, Rob_world
```

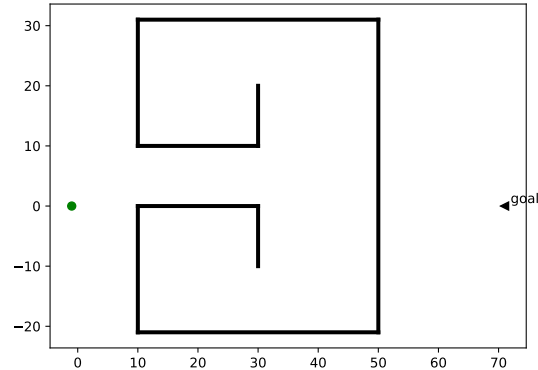


Figure 2.3: Robot trap

```

76 |
77 | world = Rob_world({((20,0),(30,20)), ((70,-5),(70,25))})
78 | body = Rob_body(world)
79 | middle = Rob_middle_layer(body)
80 | top = Rob_top_layer(middle)
81 |
82 | # try:
83 | # pl=Plot_env(body,top)
84 | # top.do({'visit':['o109','storage','o109','o103']})
85 | # You can directly control the middle layer:
86 | # middle.do({'go_to':(30,-10), 'timeout':200})
87 | # Can you make it crash?

```

Exercise 2.2 The following code implements a robot trap (Figure 2.3). It is called a trap because, once it has hit the wall, it needs to follow the wall, but local features are not enough for it to know when it can head to the goal. Write a controller that can escape the “trap” and get to the goal. See Exercise 2.4 in the textbook for hints.

```

agentTop.py — (continued)
89 | # Robot Trap for which the current controller cannot escape:
90 | trap_env = Rob_world({((10,-21),(10,0)), ((10,10),(10,31)),
91 |                      ((30,-10),(30,0)), ((30,10),(30,20)),
92 |                      ((50,-21),(50,31)), ((10,-21),(50,-21)),
93 |                      ((10,0),(30,0)), ((10,10),(30,10)),
94 |                      ((10,31),(50,31))})
95 | trap_body = Rob_body(trap_env,init_pos=(-1,0,90))
96 | trap_middle = Rob_middle_layer(trap_body)
97 | trap_top = Rob_top_layer(trap_middle,locations={'goal':(71,0)})
98 |
99 | # Robot trap exercise:
100 | # pl=Plot_env(trap_body,trap_top)
101 | # trap_top.do({'visit':['goal']})

```

Plotting for Moving Targets

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

```

agentFollowTarget.py — Plotting for moving targets
11 import matplotlib.pyplot as plt
12 from agentTop import Plot_env, body, top
13
14 class Plot_follow(Plot_env):
15     def __init__(self, body, top, epsilon=2.5):
16         """plot the agent in the environment.
17         epsilon is the threshold how close someone needs to click to
18         select a location.
19         """
20         Plot_env.__init__(self, body, top)
21         self.epsilon = epsilon
22         self.canvas = plt.gca().figure.canvas
23         self.canvas.mpl_connect('button_press_event', self.on_press)
24         self.canvas.mpl_connect('button_release_event', self.on_release)
25         self.canvas.mpl_connect('motion_notify_event', self.on_move)
26         self.pressloc = None
27         self.pressevent = None
28         for loc in self.top.locations:
29             self.display(2, f" loc {loc} at {self.top.locations[loc]}")
30
31     def on_press(self, event):
32         self.display(2, 'v', end="")
33         self.display(2, f"Press at ({event.xdata},{event.ydata}")
34         for loc in self.top.locations:
35             lx, ly = self.top.locations[loc]
36             if abs(event.xdata- lx) <= self.epsilon and abs(event.ydata-
37                 ly) <= self.epsilon :
38                 self.pressloc = loc
39                 self.pressevent = event
40                 self.display(2, "moving", loc)
41
42     def on_release(self, event):
43         self.display(2, '^', end="")
44         if self.pressloc is not None: #and event.inaxes ==
45             self.pressevent.inaxes:
46                 self.top.locations[self.pressloc] = (event.xdata, event.ydata)
47                 self.display(1, f"Placing {self.pressloc} at ({event.xdata,
48                     event.ydata})")
49         self.pressloc = None
50         self.pressevent = None
51
52     def on_move(self, event):

```

```
49         if self.pressloc is not None: # and event.inaxes ==
50             self.pressevent.inaxes:
51                 self.display(2, '-', end="")
52                 self.top.locations[self.pressloc] = (event.xdata, event.ydata)
53                 self.redraw()
54         else:
55             self.display(2, '.', end="")
56     # try:
57     # pl=Plot_follow(body, top)
58     # top.do({'visit': ['o109', 'storage', 'o109', 'o103']})
```

Exercise 2.3 Do Exercise 2.5 of Poole and Mackworth [2023].

Exercise 2.4 Change the code to also allow walls to move.

Searching for Solutions

3.1 Representing Search Problems

A search problem consists of:

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

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

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

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

```

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

```

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

```

searchProblem.py — (continued)
40 class Arc(object):
41     """An arc consists of
42         a from_node and a to_node node
43         a (non-negative) cost
44         an (optional) action
45     """
46     def __init__(self, from_node, to_node, cost=1, action=None):
47         self.from_node = from_node
48         self.to_node = to_node
49         self.cost = cost
50         assert cost >= 0, (f"Cost cannot be negative: {self}, cost={cost}")
51         self.action = action
52
53     def __repr__(self):
54         """string representation of an arc"""
55         if self.action:
56             return f"{self.from_node} --{self.action}--> {self.to_node}"
57         else:
58             return f"{self.from_node} --> {self.to_node}"

```

3.1.1 Explicit Representation of Search Graph

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

An **explicit graph** consists of

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

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

```

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

```

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

```

Graphical Display of a Search Graph

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

```

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

```

```

129         bbox =
130             dict(boxstyle="round4,pad=1.0,rounding_size=0.5",facecolor=node_color)
131         for arc in self.arcs:
132             self.show_arc(ax, arc)
133         for node in self.nodes:
134             self.show_node(ax, node, node_color = node_color)
135
136     def show_node(self, ax, node, node_color):
137         x,y = self.positions[node]
138         ax.text(x,y,node,bbox=dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
139                                     facecolor=node_color),
140                 ha='center',va='center', fontsize=self.fontsize)
141
142     def show_arc(self, ax, arc, arc_color='black', node_color='white'):
143         from_pos = self.positions[arc.from_node]
144         to_pos = self.positions[arc.to_node]
145         ax.annotate(arc.to_node, from_pos, xytext=to_pos,
146                     arrowprops={'arrowstyle': '<|-', 'linewidth': 2,
147                                   'color': arc_color},
148                     bbox=dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
149                               facecolor=node_color),
150                     ha='center',va='center',
151                     fontsize=self.fontsize)
152         # Add costs to middle of arcs:
153         if self.show_costs:
154             ax.text((from_pos[0]+to_pos[0])/2, (from_pos[1]+to_pos[1])/2,
155                     arc.cost, bbox=dict(pad=1,fc='w',ec='w'),
156                     ha='center',va='center',fontsize=self.fontsize)

```

3.1.2 Paths

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

A path is either:

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

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

searchProblem.py — (continued)

```

157 class Path(object):
158     """A path is either a node or a path followed by an arc"""
159
160     def __init__(self, initial, arc=None):
161         """initial is either a node (in which case arc is None) or
162         a path (in which case arc is an object of type Arc)"""
163         self.initial = initial
164         self.arc=arc
165         if arc is None:
166             self.cost=0
167         else:
168             self.cost = initial.cost+arc.cost
169
170     def end(self):
171         """returns the node at the end of the path"""
172         if self.arc is None:
173             return self.initial
174         else:
175             return self.arc.to_node
176
177     def nodes(self):
178         """enumerates the nodes of the path from the last element backwards
179         """
180         current = self
181         while current.arc is not None:
182             yield current.arc.to_node
183             current = current.initial
184         yield current.initial
185
186     def initial_nodes(self):
187         """enumerates the nodes for the path before the end node.
188         This calls nodes() for the initial part of the path.
189         """
190         if self.arc is not None:
191             yield from self.initial.nodes()
192
193     def __repr__(self):
194         """returns a string representation of a path"""
195         if self.arc is None:
196             return str(self.initial)
197         elif self.arc.action:
198             return f"{self.initial}\n --{self.arc.action}-->
199                 {self.arc.to_node}"
200         else:
201             return f"{self.initial} --> {self.arc.to_node}"

```

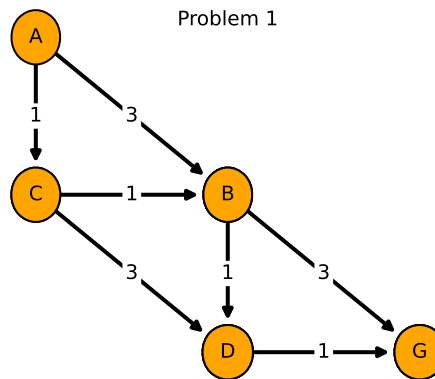


Figure 3.1: problem1

3.1.3 Example Search Problems

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

```

searchExample.py — Search Examples
11 from searchProblem import Arc, Search_problem_from_explicit_graph,
    Search_problem
12
13 problem1 = Search_problem_from_explicit_graph('Problem 1',
14     {'A', 'B', 'C', 'D', 'G'},
15     [Arc('A', 'B', 3), Arc('A', 'C', 1), Arc('B', 'D', 1), Arc('B', 'G', 3),
16       Arc('C', 'B', 1), Arc('C', 'D', 3), Arc('D', 'G', 1)],
17     start = 'A',
18     goals = {'G'},
19     positions={'A': (0, 1), 'B': (0.5, 0.5), 'C': (0, 0.5),
20               'D': (0.5, 0), 'G': (1, 0)})

```

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

```

searchExample.py — (continued)
22 problem2 = Search_problem_from_explicit_graph('Problem 2',
23     {'A', 'B', 'C', 'D', 'E', 'G', 'H', 'J'},
24     [Arc('A', 'B', 1), Arc('B', 'C', 3), Arc('B', 'D', 1), Arc('D', 'E', 3),
25       Arc('D', 'G', 1), Arc('A', 'H', 3), Arc('H', 'J', 1)],
26     start = 'A',
27     goals = {'G'},
28     positions={'A': (0, 1), 'B': (0, 3/4), 'C': (0, 0), 'D': (1/4, 3/4),
29               'E': (1/4, 0), 'G': (2/4, 3/4), 'H': (3/4, 1), 'J': (3/4, 3/4)})

```

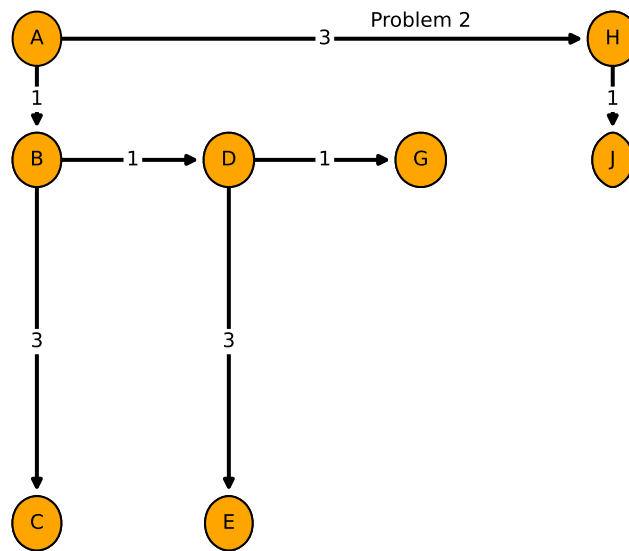


Figure 3.2: problem2

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

```

31 | searchExample.py — (continued)
32 | problem3 = Search_problem_from_explicit_graph('Problem 3',
33 |     {'a','b','c','d','e','g','h','j'},
34 |     [],
35 |     start = 'g',
36 |     goals = {'k','g'})

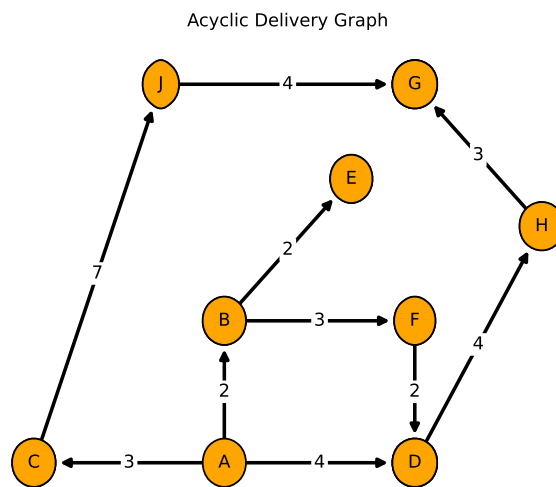
```

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

```

37 | searchExample.py — (continued)
38 | simp_delivery_graph = Search_problem_from_explicit_graph("Acyclic Delivery
39 |     Graph",
40 |     {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J'},
41 |     [
42 |         Arc('A', 'B', 2),
43 |         Arc('A', 'C', 3),
44 |         Arc('A', 'D', 4),
45 |         Arc('B', 'E', 2),
46 |         Arc('B', 'F', 3),
47 |         Arc('C', 'J', 7),
48 |         Arc('D', 'H', 4),
49 |         Arc('F', 'D', 2),

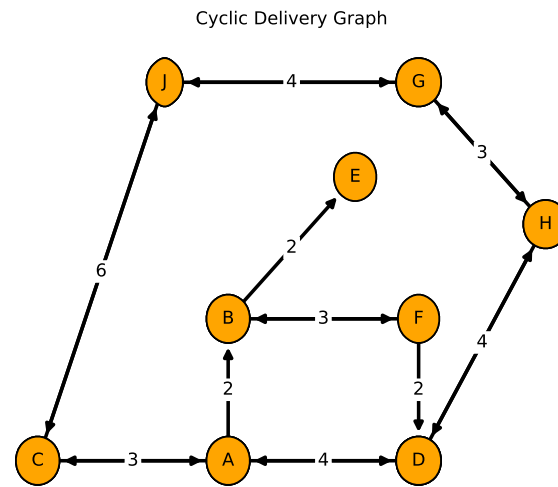
```


Figure 3.3: `simp_delivery_graph.show()`

```

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

```

Figure 3.4: `cyclic_simp_delivery_graph.show()`

73 |)

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

searchExample.py — (continued)

```

74 cyclic_simp_delivery_graph = Search_problem_from_explicit_graph("Cyclic
    Delivery Graph",
75     {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J'},
76     [
77         Arc('A', 'B', 2),
78         Arc('A', 'C', 3),
79         Arc('A', 'D', 4),
80         Arc('B', 'E', 2),
81         Arc('B', 'F', 3),
82         Arc('C', 'A', 3),
83         Arc('C', 'J', 6),
84         Arc('D', 'A', 4),
85         Arc('D', 'H', 4),
86         Arc('F', 'B', 3),
87         Arc('F', 'D', 2),
88         Arc('G', 'H', 3),
89         Arc('G', 'J', 4),
90         Arc('H', 'D', 4),
91         Arc('H', 'G', 3),
92         Arc('J', 'C', 6),
93         Arc('J', 'G', 4)],

```

```

93     start = 'A',
94     goals = {'G'},
95     hmap = {
96         'A': 7,
97         'B': 5,
98         'C': 9,
99         'D': 6,
100        'E': 3,
101        'F': 5,
102        'G': 0,
103        'H': 3,
104        'J': 4,
105    },
106    positions = {
107        'A': (0.4,0.1),
108        'B': (0.4,0.4),
109        'C': (0.1,0.1),
110        'D': (0.7,0.1),
111        'E': (0.6,0.7),
112        'F': (0.7,0.4),
113        'G': (0.7,0.9),
114        'H': (0.9,0.6),
115        'J': (0.3,0.9)
116    })

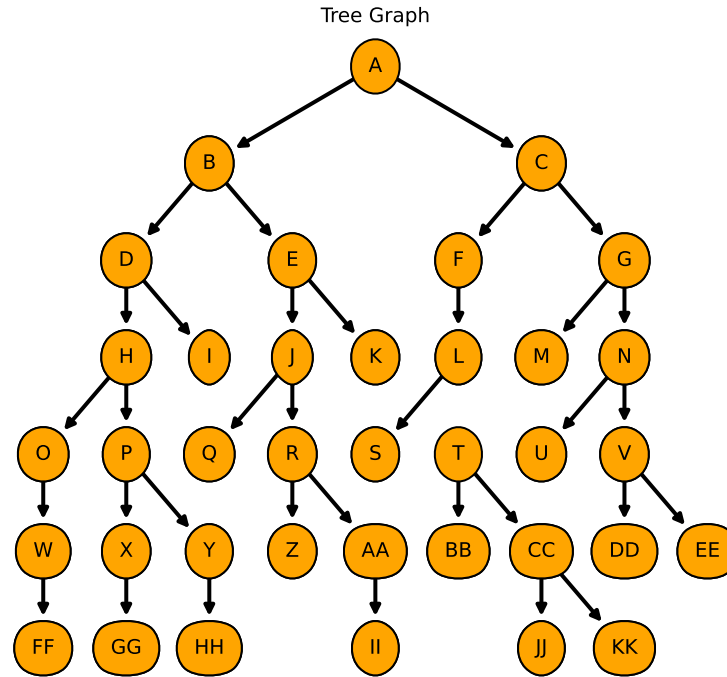
```

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

```

searchExample.py — (continued)
118 tree_graph = Search_problem_from_explicit_graph("Tree Graph",
119     {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
120      'O',
121      'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'AA', 'BB',
122      'CC',
123      'DD', 'EE', 'FF', 'GG', 'HH', 'II', 'JJ', 'KK'},
124     [
125         Arc('A', 'B', 1),
126         Arc('A', 'C', 1),
127         Arc('B', 'D', 1),
128         Arc('B', 'E', 1),
129         Arc('C', 'F', 1),
130         Arc('C', 'G', 1),
131         Arc('D', 'H', 1),
132         Arc('D', 'I', 1),
133         Arc('E', 'J', 1),
134         Arc('E', 'K', 1),
135         Arc('F', 'L', 1),
136         Arc('G', 'M', 1),
137         Arc('G', 'N', 1),
138         Arc('H', 'O', 1),
139         Arc('H', 'P', 1),
140         Arc('J', 'Q', 1),

```

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

```

138     Arc('J', 'R', 1),
139     Arc('L', 'S', 1),
140     Arc('L', 'T', 1),
141     Arc('N', 'U', 1),
142     Arc('N', 'V', 1),
143     Arc('O', 'W', 1),
144     Arc('P', 'X', 1),
145     Arc('P', 'Y', 1),
146     Arc('R', 'Z', 1),
147     Arc('R', 'AA', 1),
148     Arc('T', 'BB', 1),
149     Arc('T', 'CC', 1),
150     Arc('V', 'DD', 1),
151     Arc('V', 'EE', 1),
152     Arc('W', 'FF', 1),
153     Arc('X', 'GG', 1),
154     Arc('Y', 'HH', 1),
155     Arc('AA', 'II', 1),

```

```

156         Arc('CC', 'JJ', 1),
157         Arc('CC', 'KK', 1)
158     ],
159     start = 'A',
160     goals = {'K', 'M', 'T', 'X', 'Z', 'HH'},
161     positions = {
162         'A': (0.5,0.95),
163         'B': (0.3,0.8),
164         'C': (0.7,0.8),
165         'D': (0.2,0.65),
166         'E': (0.4,0.65),
167         'F': (0.6,0.65),
168         'G': (0.8,0.65),
169         'H': (0.2,0.5),
170         'I': (0.3,0.5),
171         'J': (0.4,0.5),
172         'K': (0.5,0.5),
173         'L': (0.6,0.5),
174         'M': (0.7,0.5),
175         'N': (0.8,0.5),
176         'O': (0.1,0.35),
177         'P': (0.2,0.35),
178         'Q': (0.3,0.35),
179         'R': (0.4,0.35),
180         'S': (0.5,0.35),
181         'T': (0.6,0.35),
182         'U': (0.7,0.35),
183         'V': (0.8,0.35),
184         'W': (0.1,0.2),
185         'X': (0.2,0.2),
186         'Y': (0.3,0.2),
187         'Z': (0.4,0.2),
188         'AA': (0.5,0.2),
189         'BB': (0.6,0.2),
190         'CC': (0.7,0.2),
191         'DD': (0.8,0.2),
192         'EE': (0.9,0.2),
193         'FF': (0.1,0.05),
194         'GG': (0.2,0.05),
195         'HH': (0.3,0.05),
196         'II': (0.5,0.05),
197         'JJ': (0.7,0.05),
198         'KK': (0.8,0.05)
199     }
200 )
201
202 # tree_graph.show(show_costs = False)

```

3.2 Generic Searcher and Variants

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

3.2.1 Searcher

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

```

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

```

```

46         self.display(1, f"Solution: {self.path} (cost:
47             {self.path.cost})\n",
48             self.num_expanded, "paths have been expanded and",
49             len(self.frontier), "paths remain in the
50             frontier")
51         return self.path
52     else:
53         self.display(4, f"Expanding: {self.path} (cost:
54             {self.path.cost})")
55         neighs = self.problem.neighbors(self.path.end())
56         self.display(2, f"Expanding: {self.path} with neighbors
57             {neighs}")
58         for arc in reversed(list(neighs)):
59             self.add_to_frontier(Path(self.path, arc))
60         self.display(3, f"New frontier: {[p.end() for p in
61             self.frontier]}")
62
63     self.display(0, "No (more) solutions. Total of",
64                 self.num_expanded, "paths expanded.")

```

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

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

```

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

```

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

3.2.2 GUI for Tracing Search

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

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

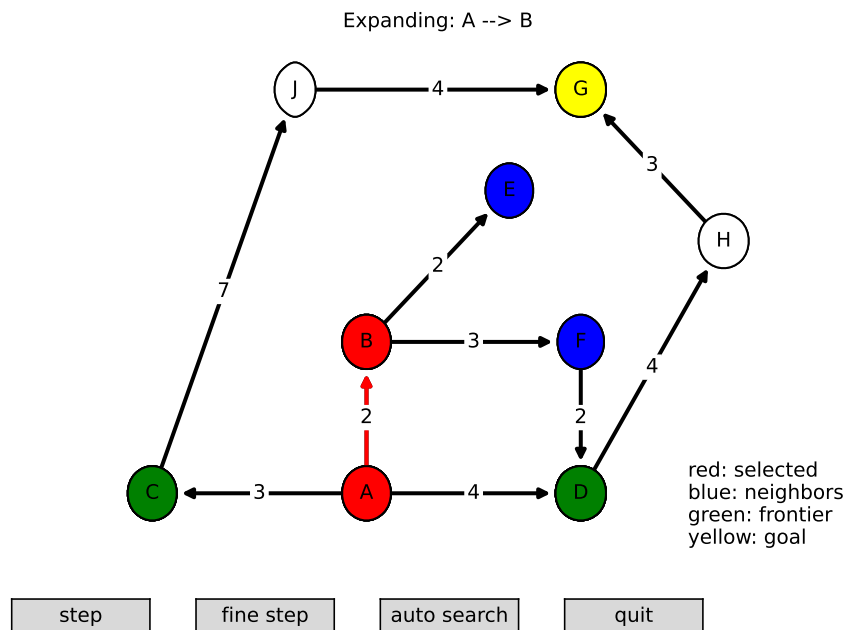


Figure 3.6: SearcherGUI(Searcher, simp.delivery_graph)

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

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

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

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

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


```

searchGUI.py — GUI for search
11 import matplotlib.pyplot as plt
12 from matplotlib.widgets import Button
13 import time
14
15 class SearcherGUI(object):
16     def __init__(self, SearchClass, problem,
17                 fontsize=10,
18                 colors = {'selected':'red', 'neighbors':'blue',
19                         'frontier':'green', 'goal':'yellow'},
20                 show_costs = True):
21         self.problem = problem
22         self.searcher = SearchClass(problem)
23         self.problem.fontsize = fontsize
24         self.colors = colors
25         self.problem.show_costs = show_costs
26         self.quitting = False
27
28         fig, self.ax = plt.subplots()
29         plt.ion() # interactive
30         self.ax.set_axis_off()
31         plt.subplots_adjust(bottom=0.15)
32         step_but = Button(plt.axes([0.05,0.02,0.15,0.05]), "step")
33         step_but.on_clicked(self.step)
34         fine_but = Button(plt.axes([0.25,0.02,0.15,0.05]), "fine step")
35         fine_but.on_clicked(self.finestep)
36         auto_but = Button(plt.axes([0.45,0.02,0.15,0.05]), "auto search")
37         auto_but.on_clicked(self.auto)
38         quit_but = Button(plt.axes([0.65,0.02,0.15,0.05]), "quit")
39         quit_but.on_clicked(self.quit)
40         self.ax.text(0.85,0, '\n'.join(self.colors[a]+": "+a
41                                     for a in self.colors))
42         self.problem.show_graph(self.ax, node_color='white')
43         self.problem.show_node(self.ax, self.problem.start,
44                               self.colors['frontier'])
45         for node in self.problem.nodes:
46             if self.problem.is_goal(node):
47                 self.problem.show_node(self.ax, node,self.colors['goal'])
48         plt.show()
49         self.click = 7 # bigger than any display!
50         self.searcher.display = self.display
51         try:
52             while self.searcher.frontier:
53                 path = self.searcher.search()
54         except ExitToPython:
55             print("Quit")
56         else:
57             print("No more solutions")
58
59     def display(self, level, *args, **nargs):

```

```

59     if self.quitting:
60         raise ExitToPython()
61     if level <= self.click: #step
62         print(*args, **nargs)
63         self.ax.set_title(f"Expanding: {self.searcher.path}",
64                           fontsize=self.problem.fontsize)
65         if level == 1:
66             self.show_frontier(self.colors['frontier'])
67             self.show_path(self.colors['selected'])
68             self.ax.set_title(f"Solution Found: {self.searcher.path}",
69                               fontsize=self.problem.fontsize)
70         elif level == 2: # what should be shown if node in multiple?
71             self.show_frontier(self.colors['frontier'])
72             self.show_path(self.colors['selected'])
73             self.show_neighbors(self.colors['neighbors'])
74         elif level == 3:
75             self.show_frontier(self.colors['frontier'])
76             self.show_path(self.colors['selected'])
77         elif level == 4:
78             self.show_frontier(self.colors['frontier'])
79
80
81     # wait for a button click
82     self.click = 0
83     plt.draw()
84     while self.click == 0 and not self.quitting:
85         plt.pause(0.1)
86         if self.quitting:
87             self.ax.set_title("Quit")
88             raise ExitToPython()
89         # undo coloring:
90         self.ax.set_title("")
91         self.show_frontier('white')
92         self.show_neighbors('white')
93         path_show = self.searcher.path
94         while path_show.arc:
95             self.problem.show_arc(self.ax, path_show.arc, 'black')
96             self.problem.show_node(self.ax, path_show.end(), 'white')
97             path_show = path_show.initial
98             self.problem.show_node(self.ax, path_show.end(), 'white')
99         if self.problem.is_goal(self.searcher.path.end()):
100             self.problem.show_node(self.ax, self.searcher.path.end(),
101                                    self.colors['goal'])
102     plt.draw()
103
104     def show_frontier(self, color):
105         for path in self.searcher.frontier:
106             self.problem.show_node(self.ax, path.end(), color)
107
108     def show_path(self, color):

```

```

109         """color selected path"""
110         path_show = self.searcher.path
111         while path_show.arc:
112             self.problem.show_arc(self.ax, path_show.arc, color)
113             self.problem.show_node(self.ax, path_show.end(), color)
114             path_show = path_show.initial
115         self.problem.show_node(self.ax, path_show.end(), color)
116
117     def show_neighbors(self, color):
118         for neigh in self.problem.neighbors(self.searcher.path.end()):
119             self.problem.show_node(self.ax, neigh.to_node, color)
120
121     def auto(self, event):
122         self.click = 1
123     def step(self, event):
124         self.click = 2
125     def finestep(self, event):
126         self.click = 3
127     def quit(self, event):
128         self.quitting = True
129
130 class ExitToPython(Exception):
131     pass

```

searchGUI.py — (continued)

```

133 from searchGeneric import Searcher, AStarSearcher
134 from searchMPP import SearcherMPP
135 import searchExample
136 from searchBranchAndBound import DF_branch_and_bound
137
138 # to demonstrate depth-first search:
139 # sdf = SearcherGUI(Searcher, searchExample.tree_graph)
140
141 # delivery graph examples:
142 # sh = SearcherGUI(Searcher, searchExample.simp_delivery_graph)
143 # sha = SearcherGUI(AStarSearcher, searchExample.simp_delivery_graph)
144 # shac = SearcherGUI(AStarSearcher,
145 #                    searchExample.cyclic_simp_delivery_graph)
146 # shm = SearcherGUI(SearcherMPP, searchExample.cyclic_simp_delivery_graph)
147 # shb = SearcherGUI(DF_branch_and_bound, searchExample.simp_delivery_graph)
148
149 # The following is AI:FCA figure 3.15, and is useful to show branch&bound:
150 # shbt = SearcherGUI(DF_branch_and_bound, searchExample.tree_graph)

```

3.2.3 Frontier as a Priority Queue

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

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

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

```

70 import heapq      # part of the Python standard library
71 from searchProblem import Path
72
73 class FrontierPQ(object):
74     """A frontier consists of a priority queue (heap), frontierpq, of
75        (value, index, path) triples, where
76        * value is the value we want to minimize (e.g., path cost + h).
77        * index is a unique index for each element
78        * path is the path on the queue
79        Note that the priority queue always returns the smallest element.
80        """
81
82     def __init__(self):
83         """constructs the frontier, initially an empty priority queue
84         """
85         self.frontier_index = 0 # the number of items added to the frontier
86         self.frontierpq = [] # the frontier priority queue
87
88     def empty(self):
89         """is True if the priority queue is empty"""
90         return self.frontierpq == []
91
92     def add(self, path, value):
93         """add a path to the priority queue
94         value is the value to be minimized"""
95         self.frontier_index += 1 # get a new unique index
96         heapq.heappush(self.frontierpq, (value, -self.frontier_index, path))
97
98     def pop(self):
99         """returns and removes the path of the frontier with minimum value.
100         """
101         (_,_,path) = heapq.heappop(self.frontierpq)
102         return path

```

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

```

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

```

3.2.4 A^* Search

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

```

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

```

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

```

searchGeneric.py — (continued)
140 import searchExample
141
142 def test(SearchClass, problem=searchExample.problem1,
          solutions=[['G', 'D', 'B', 'C', 'A']] ):

```

```

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

```

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

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

3.2.5 Multiple Path Pruning

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

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

```

searchMPP.py — Searcher with multiple-path pruning
11 from searchGeneric import AStarSearcher
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     def search(self):
23         """returns next path from an element of problem's start nodes
24         to a goal node.
25         Returns None if no path exists.
26         """
27         while not self.empty_frontier():
28             self.path = self.frontier.pop()
29             if self.path.end() not in self.explored:
30                 self.explored.add(self.path.end())
31                 self.num_expanded += 1
32                 if self.problem.is_goal(self.path.end()):
33                     self.solution = self.path # store the solution found
34                     self.display(1, f"Solution: {self.path} (cost:
35                               {self.path.cost})\n",
36                               self.num_expanded, "paths have been expanded and",
37                               len(self.frontier), "paths remain in the
38                               frontier")
39                     return self.path
40                 else:
41                     self.display(4, f"Expanding: {self.path} (cost:
42                               {self.path.cost})")
43                     neighs = self.problem.neighbors(self.path.end())
44                     self.display(2, f"Expanding: {self.path} with neighbors
45                               {neighs}")
46                     for arc in neighs:
47                         self.add_to_frontier(Path(self.path, arc))
48                     self.display(3, f"New frontier: {[p.end() for p in
49                               self.frontier]}")
50         self.display(0, "No (more) solutions. Total of",

```

```

46         self.num_expanded, "paths expanded.")
47
48 from searchGeneric import test
49 if __name__ == "__main__":
50     test(SearcherMPP)
51
52 import searchExample
53 # searcherMPPcdp = SearcherMPP(searchExample.cyclic_simp_delivery_graph)
54 # searcherMPPcdp.search() # find first path
55
56 # To use the GUI for SearcherMPP do
57 # python -i searchGUI.py
58 # import searchMPP
59 # SearcherGUI(searchMPP.SearcherMPP,
    searchExample.cyclic_simp_delivery_graph)

```

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

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

```

searchGrid.py — A grid problem to demonstrate A*
11 from searchProblem import Search_problem, Arc
12
13 class GridProblem(Search_problem):
14     """a node is a pair (x,y)"""
15     def __init__(self, size=10):
16         self.size = size
17
18     def start_node(self):
19         """returns the start node"""
20         return (0,0)
21
22     def is_goal(self, node):
23         """returns True when node is a goal node"""
24         return node == (self.size, self.size)
25
26     def neighbors(self, node):
27         """returns a list of the neighbors of node"""
28         (x,y) = node
29         return [Arc(node, (x+1,y)), Arc(node, (x,y+1))]
30
31     def heuristic(self, node):
32         (x,y) = node

```



```

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

```

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

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

3.3 Branch-and-bound Search

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

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

searchBranchAndBound.py — Branch and Bound Search

```

11 from searchProblem import Path

```

```

12 from searchGeneric import Searcher
13 from display import Displayable
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     def search(self):
32         """returns an optimal solution to a problem with cost less than
33         bound.
34         returns None if there is no solution with cost less than bound."""
35         self.frontier = [Path(self.problem.start_node())]
36         self.num_expanded = 0
37         while self.frontier:
38             self.path = self.frontier.pop()
39             if self.path.cost+self.problem.heuristic(self.path.end()) <
40                 self.bound:
41                 # if self.path.end() not in self.path.initial_nodes(): # for
42                 # cycle pruning
43                 self.display(2,"Expanding:",self.path,"cost:",self.path.cost)
44                 self.num_expanded += 1
45                 if self.problem.is_goal(self.path.end()):
46                     self.best_path = self.path
47                     self.bound = self.path.cost
48                     self.display(1,"New best path:",self.path,"
49                     cost:",self.path.cost)
50                 else:
51                     neighs = self.problem.neighbors(self.path.end())
52                     self.display(4,"Neighbors are", neighs)
53                     for arc in reversed(list(neighs)):
54                         self.add_to_frontier(Path(self.path, arc))
55                     self.display(3, f"New frontier: {[p.end() for p in
56                     self.frontier]}")
57         self.path = self.best_path
58         self.solution = self.best_path
59         self.display(1,f"Optimal solution is {self.best_path}." if
60             self.best_path
61             else "No solution found.",

```

```

53         f"Number of paths expanded: {self.num_expanded}."
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) _____
56 from searchGeneric import test
57 if __name__ == "__main__":
58     test(DF_branch_and_bound)
59
60 # Example queries:
61 import searchExample
62 # searcher1 = DF_branch_and_bound(searchExample.simp_delivery_graph)
63 # searcher1.search()      # find optimal path
64 # searcher2 =
65     DF_branch_and_bound(searchExample.cyclic_simp_delivery_graph,
66                           bound=100)
67 # searcher2.search()      # find optimal path
68
69 # to use the GUI do:
70 # ipython -i searchGUI.py
71 # import searchBranchAndBound
72 # SearcherGUI(searchBranchAndBound.DF_branch_and_bound,
73               searchExample.simp_delivery_graph)
74 # SearcherGUI(searchBranchAndBound.DF_branch_and_bound,
75               searchExample.cyclic_simp_delivery_graph)

```

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

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

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

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

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

```

_____searchTest.py — code that may be useful to compare A* and branch-and-bound _____
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*1.00001
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*10+10
43     print("\nBranch and bound (with not-very-good initial bound of",
44           bbound, ")")
45     tbb2 = DF_branch_and_bound(problem,bbound)
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)")
51     tsearcher = Searcher(problem)
52     print("Path found:",tsearcher.search()," cost=",tsearcher.solution.cost)
53
54 import searchExample
55 from searchTest import run
56 if __name__ == "__main__":
57     run(searchExample.problem1,"Problem 1")
58     # run(searchExample.simp_delivery_graph,"Acyclic Delivery")
59     # run(searchExample.cyclic_simp_delivery_graph,"Cyclic Delivery")
60 # 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 matters for some algorithms.

```
variable.py — Representations of a variable in CSPs and probabilistic models
11 import random
12
13 class Variable(object):
14     """A random variable.
15     name (string) - name of the variable
16     domain (list) - a list of the values for the variable.
17     an (x,y) position for displaying
18     """
19
20     def __init__(self, name, domain, position=None):
21         """Variable
22         name a string
23         domain a list of printable values
24         position of form (x,y) where 0 <= x <= 1, 0 <= y <= 1
25         """
26         self.name = name # string
27         self.domain = domain # list of values
28         self.position = position if position else (random.random(),
29                                                    random.random())
29         self.size = len(domain)
30
31     def __str__(self):
```

```

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

```

4.1.2 Constraints

A **constraint** consists of:

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

```

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

```

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

If con is a constraint:

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

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

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

```

cspProblem.py — (continued)
32 def can_evaluate(self, assignment):
33     """
34     assignment is a variable:value dictionary
35     returns True if the constraint can be evaluated given assignment
36     """
37     return all(v in assignment for v in self.scope)
38
39 def holds(self, assignment):
40     """returns the value of Constraint con evaluated in assignment.
41
42     precondition: all variables are assigned in assignment, ie
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:

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

Other properties are inferred from these:

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

```

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

```

```

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 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). Unless the assignment assigns to all variables, `consistent` does *not* imply the CSP is consistent or has a solution, because constraints involving variables not in the assignment are ignored.

cspProblem.py — (continued)

```

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. This also includes code used for the consistency GUI (Section 4.4.2).

cspProblem.py — (continued)

```

83     def show(self, linewidth=3, showDomains=False, showAutoAC = False):
84         self.linewidth = linewidth
85         self.picked = None
86         plt.ion() # interactive
87         self.arcs = {} # arc: (con,var) dictionary
88         self.thelines = {} # (con,var):arc dictionary
89         self.nodes = {} # node: variable dictionary

```

```

90     self.fig, self.ax= plt.subplots(1, 1)
91     self.ax.set_axis_off()
92     for var in self.variables:
93         if var.position is None:
94             var.position = (random.random(), random.random())
95     self.showAutoAC = showAutoAC # used for consistency GUI
96     self.autoAC = False
97     domains = {var:var.domain for var in self.variables} if showDomains
98         else {}
99     self.draw_graph(domains=domains)
100
101 def draw_graph(self, domains={}, to_do = {}, title=None, fontsize=10):
102     self.ax.clear()
103     self.ax.set_axis_off()
104     if title:
105         plt.title(title, fontsize=fontsize)
106     else:
107         plt.title(self.title, fontsize=fontsize)
108     var_bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
109                     facecolor="yellow")
110     con_bbox = dict(boxstyle="square,pad=1.0",facecolor="lightyellow")
111     self.autoACtext = plt.text(0,0,"Auto AC" if self.showAutoAC else "",
112                               bbox={'boxstyle':'square,pad=1.0',
113                                     'facecolor':'pink'},
114                               picker=True, fontsize=fontsize)
115
116     for con in self.constraints:
117         if con.position is None:
118             con.position = tuple(sum(var.position[i] for var in
119                                     con.scope)/len(con.scope)
120                                for i in range(2))
121         cx,cy = con.position
122         bbox = con_bbox
123         for var in con.scope:
124             vx,vy = var.position
125             if (var,con) in to_do:
126                 color = 'blue'
127             else:
128                 color = 'green'
129             line = lines.Line2D([cx,vx], [cy,vy], axes=self.ax,
130                                color=color,
131                                picker=True, pickradius=10,
132                                linewidth=self.linewidth)
133             self.arcs[line]= (var,con)
134             self.thelines[(var,con)] = line
135             self.ax.add_line(line)
136         plt.text(cx,cy,con.string,
137                 bbox=con_bbox,
138                 ha='center',va='center', fontsize=fontsize)
139
140     for var in self.variables:
141         x,y = var.position

```

```

136         if domains:
137             node_label = f"{var.name}\n{domains[var]}"
138         else:
139             node_label = var.name
140         node = plt.text(x, y, node_label, bbox=var_bbox, ha='center',
141                        va='center',
142                        picker=True, fontsize=fontsize)
143         self.nodes[node] = var
144     self.fig.canvas.mpl_connect('pick_event', self.pick_handler)

```

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

```

cspProblem.py — (continued)
145 def pick_handler(self,event):
146     mouseevent = event.mouseevent
147     self.last_artist = artist = event.artist
148     #print('***picker handler:',artist, 'mouseevent:', mouseevent)
149     if artist in self.arcs:
150         #print('### selected arc',self.arcs[artist])
151         self.picked = self.arcs[artist]
152     elif artist in self.nodes:
153         #print('### selected node',self.nodes[artist])
154         self.picked = self.nodes[artist]
155     elif artist==self.autoACtext:
156         self.autoAC = True
157         #print("*** autoAC")
158     else:
159         print("### unknown click")

```

4.1.4 Examples

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

```

cspExamples.py — Example CSPs
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     # return lambda x: x != val # alternative definition
17     # return partial(ne,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 |     # return lambda x: x == val # alternative definition
26 |     # return partial(eq,val) # another alternative definition
27 |     def isv(x):
28 |         return val == x
29 |     isv.__name__ = f"{val} == "
30 |     return isv

```

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

```

_____cspExamples.py — (continued)_____
32 | X = Variable('X', {1,2,3}, position=(0.1,0.8))
33 | Y = Variable('Y', {1,2,3}, position=(0.5,0.2))
34 | Z = Variable('Z', {1,2,3}, position=(0.9,0.8))
35 | csp0 = CSP("csp0", {X,Y,Z},
36 |           [ Constraint([X,Y], lt, "X<Y"),
37 |             Constraint([Y,Z], lt, "Y<Z")])

```

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

```

_____cspExamples.py — (continued)_____
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=(1,0.3))
42 | C0 = Constraint([A,B], lt, "A < B", position=(0.4,0.3))
43 | C1 = Constraint([B], ne_(2), "B != 2", position=(1,0.7))
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

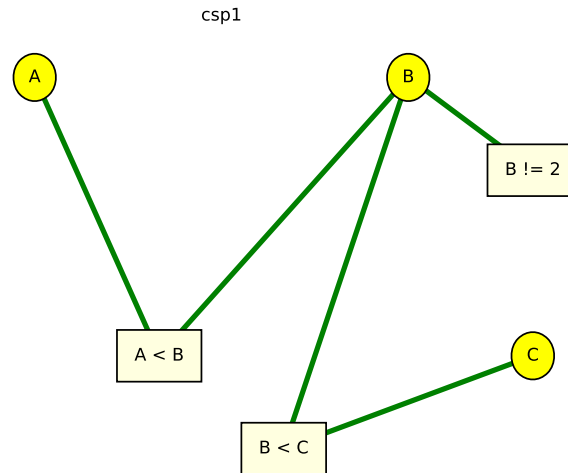
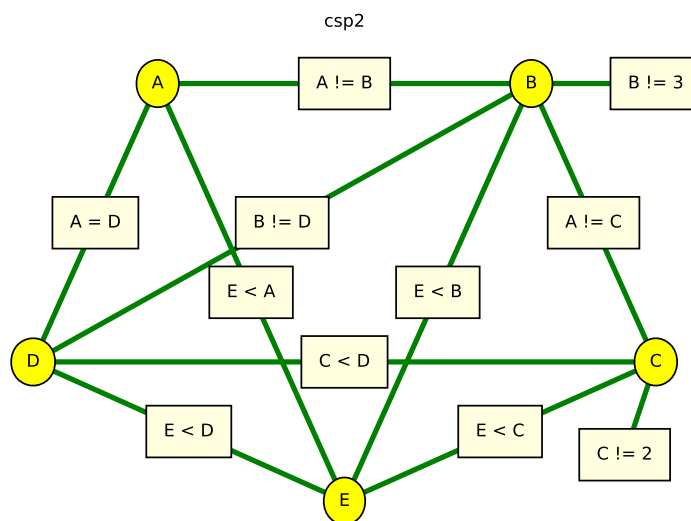
```

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 two more.

```

_____cspExamples.py — (continued)_____
51 | D = Variable('D', {1,2,3,4}, position=(0,0.3))

```

Figure 4.1: `csp1.show()`Figure 4.2: `csp2.show()`

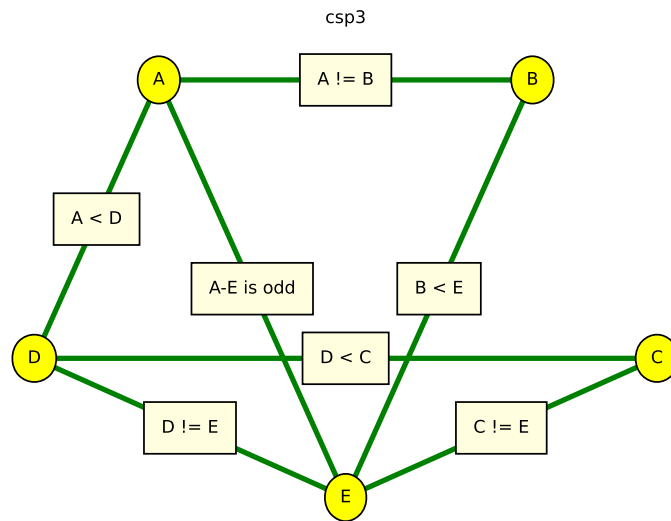


Figure 4.3: csp3.show()

```

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=(0.95,0.1)),
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")])

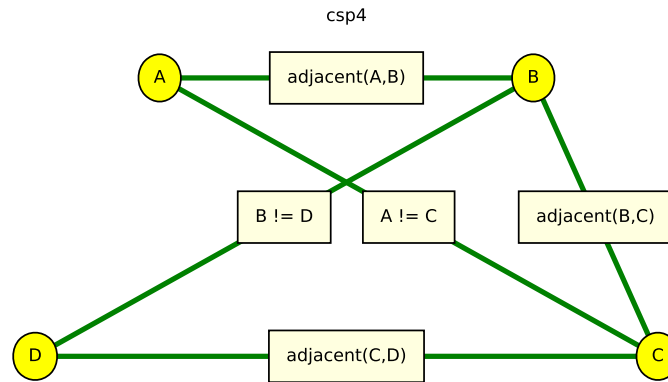
```

The following example is another scheduling problem (but with multiple answers). This is the same as “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"),
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")])

```

Figure 4.4: `csp4.show()`

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

```

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

```

The following examples represent the crossword shown in Figure 4.5.

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

```

86 def meet_at(p1,p2):
87     """returns a function of two words that is true
88         when the words intersect at positions p1, p2.

```

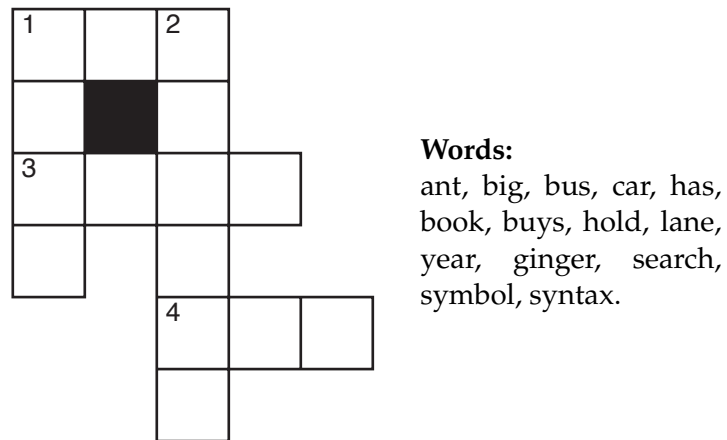



Figure 4.5: crossword1: a crossword puzzle to be solved

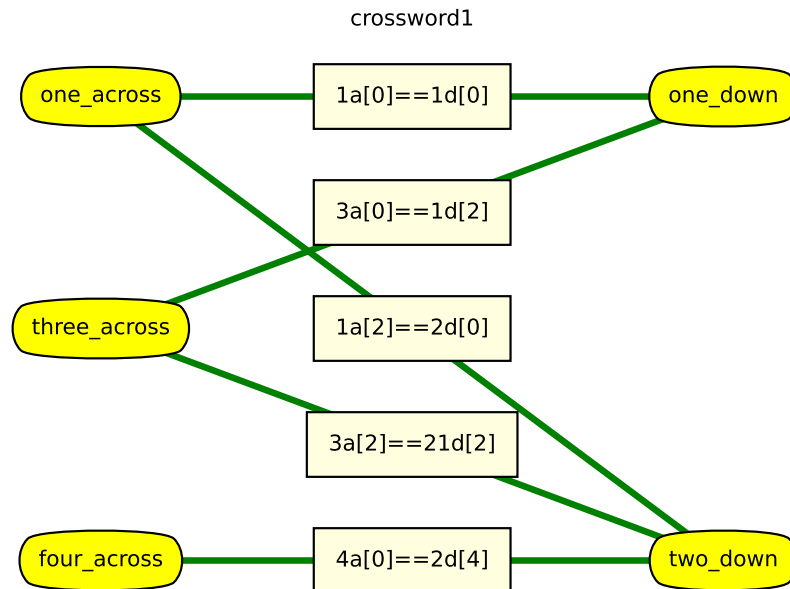


Figure 4.6: crossword1.show()

```

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

```

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

```

cspExamples.py — (continued) —
112 words = {'ant', 'big', 'bus', 'car', 'has', 'book', 'buys', 'hold',
113         'lane', 'year', 'ginger', 'search', 'symbol', 'syntax'}
114
115 def is_word(*letters, words=words):
116     """is true if the letters concatenated form a word in words"""
117     return "".join(letters) in words
118
119 letters = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
120         "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y",
121         "z"}
122

```

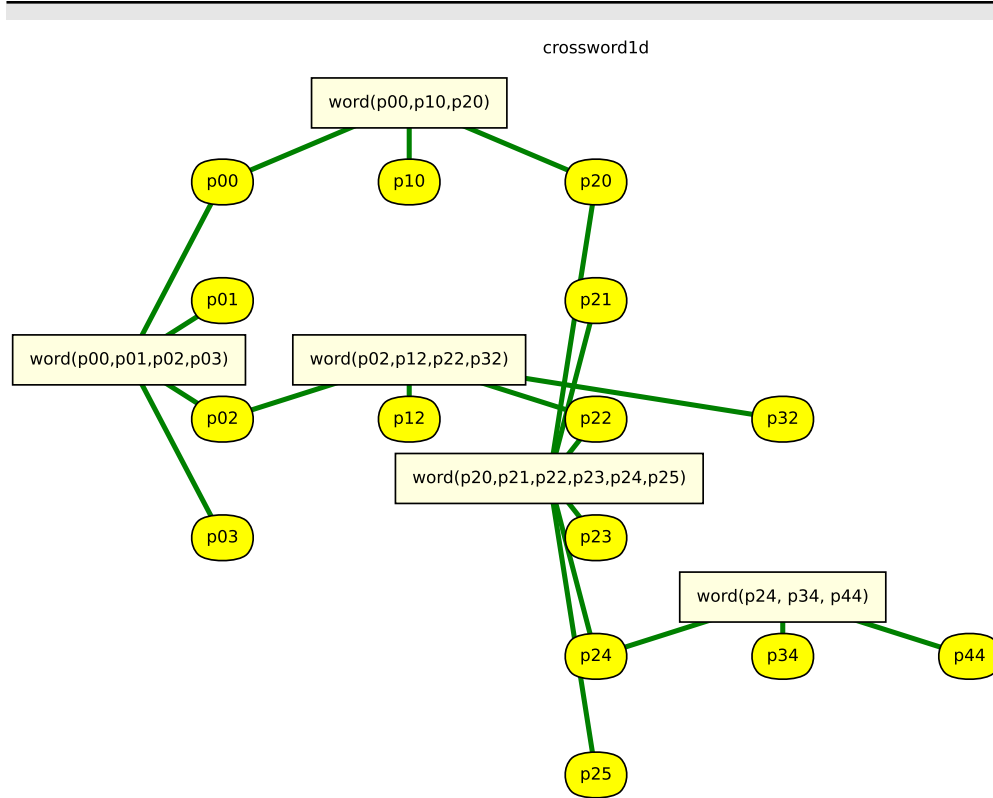


Figure 4.7: crossword1d.show()

```

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

```

```

143         p02, p12, p22, p32, # third row
144         p03, p23, #fourth row
145         p24, p34, p44, # fifth row
146         p25 # sixth row
147     },
148     [Constraint([p00, p10, p20], is_word, "word(p00,p10,p20)",
149                position=(0.3,0.95)), #1-across
150     Constraint([p00, p01, p02, p03], is_word,
151                "word(p00,p01,p02,p03)",
152                position=(0,0.625)), # 1-down
153     Constraint([p02, p12, p22, p32], is_word,
154                "word(p02,p12,p22,p32)",
155                position=(0.3,0.625)), # 3-across
156     Constraint([p20, p21, p22, p23, p24, p25], is_word,
157                "word(p20,p21,p22,p23,p24,p25)",
158                position=(0.45,0.475)), # 2-down
159     Constraint([p24, p34, p44], is_word, "word(p24, p34,
160                p44)",
161                position=(0.7,0.325)) # 4-across
162 ])
```

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)
160 def queens(ri,rj):
161     """ri and rj are different rows, return the condition that the queens
162         cannot take each other"""
163     def no_take(ci,cj):
164         """is true if queen at (ri,ci) cannot take a queen at (rj,cj)"""
165         return ci != cj and abs(ri-ci) != abs(rj-cj)
166     return no_take
167 def n_queens(n):
168     """returns a CSP for n-queens"""
169     columns = list(range(n))
170     variables = [Variable(f"R{i}",columns) for i in range(n)]
171     # note positions will be random
172     return CSP("n-queens",
173               variables,
```

```

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

```

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

Unit tests

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

```

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

```

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

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

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

4.2 A Simple Depth-first Solver

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

```

cspDFS.py — Solving a CSP using depth-first search.
11 import cspExamples
12
13 def dfs_solver(constraints, context, var_order):

```

```

14     """generator for all solutions to csp.
15     context is an assignment of values to some of the variables.
16     var_order is a list of the variables in csp that are not in context.
17     """
18     to_eval = {c for c in constraints if c.can_evaluate(context)}
19     if all(c.holds(context) for c in to_eval):
20         if var_order == []:
21             yield context
22         else:
23             rem_cons = [c for c in constraints if c not in to_eval]
24             var = var_order[0]
25             for val in var.domain:
26                 yield from dfs_solver(rem_cons, context|{var:val},
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"""
38     if var_order == None: # use an arbitrary variable order
39         var_order = list(csp.variables)
40     for sol in dfs_solver(csp.constraints, {}, var_order):
41         return sol #return first one
42
43 if __name__ == "__main__":
44     cspExamples.test_csp(dfs_solve1)
45
46 #Try:
47 # dfs_solve_all(cspExamples.csp1)
48 # dfs_solve_all(cspExamples.csp2)
49 # dfs_solve_all(cspExamples.crossword1)
50 # dfs_solve_all(cspExamples.crossword1d) # warning: may take a *very* long
51     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 import cspExamples
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(cspExamples.csp1))
64 #print(searcher1.search()) # get next solution
65 searcher2 = Searcher(Search_from_CSP(cspExamples.csp2))
66 #print(searcher2.search()) # get next solution
67 searcher3 = Searcher(Search_from_CSP(cspExamples.crossword1))
68 #print(searcher3.search()) # get next solution
69 searcher4 = Searcher(Search_from_CSP(cspExamples.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):
17         """a CSP solver that uses arc consistency
18         * csp is the CSP to be solved
19         """
20         self.csp = csp

```

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

```

_____cspConsistency.py — (continued) _____
22 def make_arc_consistent(self, domains=None, to_do=None):
23     """Makes this CSP arc-consistent using generalized arc consistency
24     domains is a variable:domain dictionary
25     to_do is a set of (variable,constraint) pairs
26     returns the reduced domains (an arc-consistent variable:domain
27         dictionary)
28     """
29     if domains is None:
30         self.domains = {var:var.domain for var in self.csp.variables}
31     else:
32         self.domains = domains.copy() # use a copy of domains
33     if to_do is None:
34         to_do = {(var, const) for const in self.csp.constraints
35                 for var in const.scope}
36     else:
37         to_do = to_do.copy() # use a copy of to_do
38     self.display(5, "Performing AC with domains", self.domains)
39     while to_do:
40         self.arc_selected = (var, const) = self.select_arc(to_do)

```

```

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

```

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

```

_cspConsistency.py — (continued)
65     def select_arc(self, to_do):
66         """Selects the arc to be taken from to_do .
67         * to_do is a set of arcs, where an arc is a (variable,constraint)
68         pair
69         the element selected must be removed from to_do.
70         """
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 treats unary (with no other variables in the constraint) and binary (with one other variables in the constraint) constraints as special cases. These cases are not strictly necessary; the last case covers the first two cases, but is more difficult to understand without seeing the first two cases. Note that this case analysis is not in the code distribution, but can replace the assignment to *new_domain* above.

```

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

```

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

```

4.4.1 Direct Implementation of Domain Splitting

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

```

_____cspConsistency.py — (continued)_____
86 def generate_sols(self, domains=None, to_do=None, context=dict()):
87     """return list of all solution to the current CSP
88     to_do is the list of arcs to check
89     context is a dictionary of splits made (used for display)
90     """
91     new_domains = self.make_arc_consistent(domains, to_do)

```

```

92     if any(len(new_domains[var]) == 0 for var in new_domains):
93         self.display(1, f"No solutions for context {context}")
94     elif all(len(new_domains[var]) == 1 for var in new_domains):
95         self.display(1, "solution:", str({var: select(
96             new_domains[var] for var in new_domains}))
97         yield {var: select(new_domains[var]) for var in new_domains}
98     else:
99         var = self.select_var(x for x in self.csp.variables if
100                               len(new_domains[x]) > 1)
101         dom1, dom2 = partition_domain(new_domains[var])
102         self.display(5, "...splitting", var, "into", dom1, "and", dom2)
103         new_doms1 = new_domains | {var: dom1}
104         new_doms2 = new_domains | {var: dom2}
105         to_do = self.new_to_do(var, None)
106         self.display(4, " adding", to_do if to_do else "nothing", "to
107             to_do.")
108         yield from self.generate_sols(new_doms1, to_do,
109             context|{var: dom1})
110         yield from self.generate_sols(new_doms2, to_do,
111             context|{var: dom1})
112
113     def solve_all(self, domains=None, to_do=None):
114         return list(self.generate_sols())
115
116     def solve_one(self, domains=None, to_do=None):
117         return select(self.generate_sols())
118
119     def select_var(self, iter_vars):
120         """return the next variable to split"""
121         return select(iter_vars)
122
123     def partition_domain(dom):
124         """partitions domain dom into two.
125         """
126         split = len(dom) // 2
127         dom1 = set(list(dom)[:split])
128         dom2 = dom - dom1
129         return dom1, dom2

```

cspConsistency.py — (continued)

```

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

```

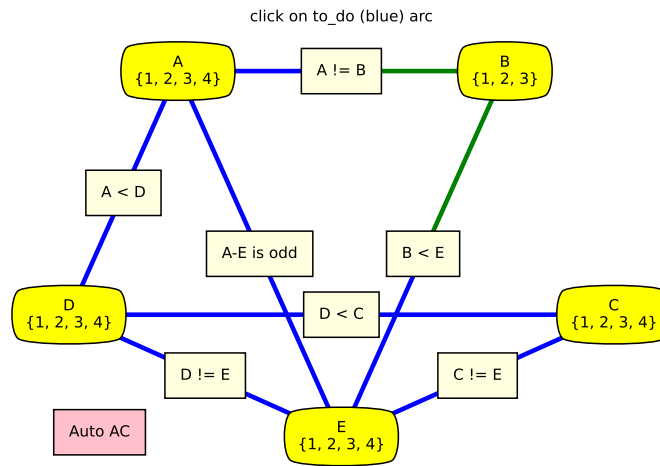


Figure 4.8: ConsistencyGUI(cspExamples.csp3).go()

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

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

Unit test:

```

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

```

4.4.2 Consistency GUI

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

Figure 4.8 shows the state of the GUI after two arcs have been made arc consistent. The arcs on the to_do list are colored blue. The green arcs are those that have been made arc consistent. The user can click on a blue arc to process

that arc. If the arc selected is not arc consistent, it is made red, the domain is reduced, and then the arc becomes green. If the arc was already arc consistent it turns green.

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

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

```

_____cspConsistencyGUI.py — GUI for consistency-based CSP solving_____
11 from cspConsistency import Con_solver
12 import matplotlib.pyplot as plt
13
14 class ConsistencyGUI(Con_solver):
15     def __init__(self, csp, fontsize=10, speed=1, **kwargs):
16         """
17         csp is the csp to show
18         fontsize is the size of the text
19         speed is the number of animations per second (controls delay_time)
20             1 (slow) and 4 (fast) seem like good values
21         """
22         self.fontsize = fontsize
23         self.delay_time = 1/speed
24         Con_solver.__init__(self, csp, **kwargs)
25         csp.show(showAutoAC = True)
26
27     def go(self):
28         res = self.solve_all()
29         self.csp.draw_graph(domains=self.domains,
30                             title="No more solutions. GUI finished. ",
31                             fontsize=self.fontsize)
32         return res
33
34     def select_arc(self, to_do):
35         while True:
36             self.csp.draw_graph(domains=self.domains, to_do=to_do,
37                                 title="click on to_do (blue) arc",
38                                 fontsize=self.fontsize)
39             while self.csp.picked == None and not self.csp.autoAC:
40                 plt.pause(0.01) # controls reaction time of GUI
41             if self.csp.autoAC:
42                 break
43             picked = self.csp.picked
44             self.csp.picked = None
45             if picked in to_do:
46                 to_do.remove(picked)
47                 print(f"{picked} picked")
48                 return picked
49             else:

```

```

49         print(f"{picked} not in to_do")
50     if self.csp.autoAC:
51         self.csp.draw_graph(domains=self.domains, to_do=to_do,
52                             title="Auto AC", fontsize=self.fontsize)
53         plt.pause(self.delay_time)
54         return to_do.pop()
55
56     def select_var(self, iter_vars):
57         vars = list(iter_vars)
58         while True:
59             self.csp.draw_graph(domains=self.domains,
60                                 title="Arc consistent. Click node to
61                                     split",
62                                 fontsize=self.fontsize)
63             while self.csp.picked == None:
64                 plt.pause(0.01) # controls reaction time of GUI
65                 picked = self.csp.picked
66                 self.csp.picked = None
67                 self.csp.autoAC = False
68                 if picked in vars:
69                     #print("splitting",picked)
70                     return picked
71                 else:
72                     print(picked,"not in",vars)
73
74     def display(self,n,*args,**nargs):
75         if n <= self.max_display_level: # default display
76             print(*args, **nargs)
77         if n==1: # solution found or no solutions"
78             self.csp.draw_graph(domains=self.domains, to_do=set(),
79                                 title=' '.join(args)+" : click any node or
80                                     arc to continue",
81                                 fontsize=self.fontsize)
82             self.csp.autoAC = False
83             while self.csp.picked == None and not self.csp.autoAC:
84                 plt.pause(0.01) # controls reaction time of GUI
85                 self.csp.picked = None
86             elif n==2: # backtracking
87                 plt.title("backtracking: click any node or arc to continue")
88                 self.csp.autoAC = False
89                 while self.csp.picked == None and not self.csp.autoAC:
90                     plt.pause(0.01)
91                     self.csp.picked = None
92             elif n==3: # inconsistent arc
93                 line = self.csp.thelines[self.arc_selected]
94                 line.set_color('red')
95                 line.set_linewidth(10)
96                 plt.pause(self.delay_time)
97                 line.set_color('limegreen')
98                 line.set_linewidth(self.csp.linewidth)

```

```

97         #elif n==4 and self.add_to_do: # adding to to_do
98         #     print("adding to to_do",self.add_to_do) ## highlight these arc
99
100     import cspExamples
101     # Try:
102     # ConsistencyGUI(cspExamples.csp1).go()
103     # ConsistencyGUI(cspExamples.csp3).go()
104     # ConsistencyGUI(cspExamples.csp3, speed=4, fontsize=15).go()

```

4.4.3 Domain Splitting as an interface to graph searching

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

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

```

cspConsistency.py — (continued)
147 from searchProblem import Arc, Search_problem
148
149 class Search_with_AC_from_CSP(Search_problem, Displayable):
150     """A search problem with arc consistency and domain splitting
151
152     A node is a CSP """
153     def __init__(self, csp):
154         self.cons = Con_solver(csp) #copy of the CSP
155         self.domains = self.cons.make_arc_consistent()
156
157     def is_goal(self, node):
158         """node is a goal if all domains have 1 element"""
159         return all(len(node[var])==1 for var in node)
160
161     def start_node(self):
162         return self.domains
163
164     def neighbors(self, node):
165         """returns the neighboring nodes of node.
166         """
167         neighs = []
168         var = select(x for x in node if len(node[x])>1)
169         if var:
170             dom1, dom2 = partition_domain(node[var])
171             self.display(2, "Splitting", var, "into", dom1, "and", dom2)
172             to_do = self.cons.new_to_do(var, None)
173             for dom in [dom1, dom2]:
174                 newdoms = node | {var: dom}
175                 cons_doms = self.cons.make_arc_consistent(newdoms, to_do)
176                 if all(len(cons_doms[v])>0 for v in cons_doms):
177                     # all domains are non-empty
178                     neighs.append(Arc(node, cons_doms))
179             else:

```



```

180         self.display(2, "...", var, "in", dom, "has no solution")
181     return neighs

```

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

Unit test:

```

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

```

Testing:

```

cspConsistency.py — (continued)
195 ## Test Solving CSPs with Arc consistency and domain splitting:
196 #Con_solver.max_display_level = 4 # display details of AC (0 turns off)
197 #Con_solver(cspExamples.csp1).solve_all()
198 #searcher1d = Searcher(Search_with_AC_from_CSP(cspExamples.csp1))
199 #print(searcher1d.search())
200 #Searcher.max_display_level = 2 # display search trace (0 turns off)
201 #searcher2c = Searcher(Search_with_AC_from_CSP(cspExamples.csp2))
202 #print(searcher2c.search())
203 #searcher3c = Searcher(Search_with_AC_from_CSP(cspExamples.crossword1))
204 #print(searcher3c.search())
205 #searcher4c = Searcher(Search_with_AC_from_CSP(cspExamples.crossword1d))
206 #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

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

cspSLS.py — (continued)

```

63  def search_with_any_conflict(self, max_steps, prob_anycon=1.0):
64      """Searches with the any_conflict heuristic.
65      This relies on just maintaining the set of conflicts;
66      it does not maintain a priority queue
67      """
68      self.variable_pq = None # we are not maintaining the priority queue.
69                          # This ensures it is regenerated if
70                          # we call search_with_var_pq.
71      for i in range(max_steps):
72          self.number_of_steps +=1
73          if random.random() < prob_anycon:
74              con = random.choice(self.conflicts) # pick random conflict
75              var = random.choice(con.scope) # pick variable in conflict
76          else:
77              var = random.choice(self.variables_to_select)
78          if len(var.domain) > 1:
79              val = random.choice([val for val in var.domain
80                              if val is not
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 value for the variable selected. Modify the code to include an option selects a value for the selected variable that reduces the number of conflicts the most. Have a parameter that specifies the probability that the best value is chosen, and otherwise chooses a value at random.

4.5.2 Two-Stage Choice

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

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

```

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

```

```

129         self.display(3, "Became consistent", varcon)
130         self.conflicts.remove(varcon)
131         for v in varcon.scope: # v is in one fewer
            conflicts
132             var_differential[v] =
                var_differential.get(v, 0) - 1
133         else:
134             if varcon not in self.conflicts: # was consis, not now
135                 self.display(3, "Became inconsistent", varcon)
136                 self.conflicts.add(varcon)
137                 for v in varcon.scope: # v is in one more
                    conflicts
138                     var_differential[v] =
                        var_differential.get(v, 0) + 1
139             self.variable_pq.update_each_priority(var_differential)
140             self.display(2, "Number of conflicts", len(self.conflicts))
141             if not self.conflicts: # no conflicts, so solution found
142                 self.display(1, "Solution found:",
                    self.current_assignment, "in",
143                     self.number_of_steps, "steps")
144             return self.number_of_steps
145         self.display(1, "No solution in", self.number_of_steps, "steps",
146                     len(self.conflicts), "conflicts remain")
147         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.

```

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

```

```

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

```

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

4.5.3 Updatable Priority Queues

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

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

```

cspSLS.py — (continued)
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             triple = self.pq[0]
233         return triple[2], triple[0] # elt, value
234
235     def empty(self):
236         """returns True iff the priority queue is empty"""
237         return all(triple[2] == self.REMOVED for triple in self.pq)

```


4.5.4 Plotting Run-Time Distributions

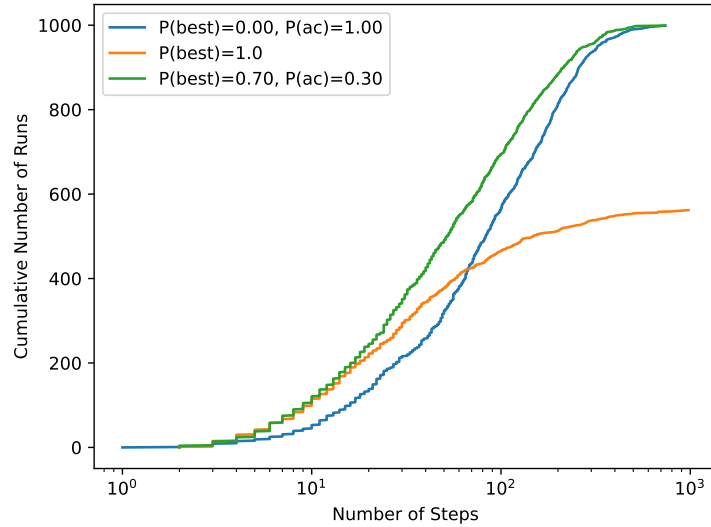
Runtime_distribution uses matplotlib to plot run time distributions. Here the run time is a misnomer as we are only plotting the number of steps, not the time. Computing the run time is non-trivial as many of the runs have a very short run time. To compute the time accurately would require running the 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 4.9 gives run-time distributions for 3 algorithms. It is also useful to compare the distributions of different runs of the same algorithms and settings.

Figure 4.9: Run-time distributions for three algorithms on *csp2*.

4.5.5 Testing

```

cspSLS.py — (continued)
273 import cspExamples
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     cspExamples.test_csp(sls_solver)
285     cspExamples.test_csp(any_conflict_solver)
286
287 ## Test Solving CSPs with Search:
288 #se1 = SLSearcher(cspExamples.csp1); print(se1.search(100))
289 #se2 = SLSearcher(cspExamples.csp2); print(se2.search(1000,1.0)) # greedy
290 #se2 = SLSearcher(cspExamples.csp2); print(se2.search(1000,0)) #
    any_conflict
291 #se2 = SLSearcher(cspExamples.csp2); print(se2.search(1000,0.7)) # 70%
    greedy; 30% any_conflict
292 #SLSearcher.max_display_level=2 #more detailed display
293 #se3 = SLSearcher(cspExamples.crossword1); print(se3.search(100),0.7)

```

```

294 #p = Runtime_distribution(cspExamples.csp2)
295 #p.plot_runs(1000,1000,0) # any_conflict
296 #p.plot_runs(1000,1000,1.0) # greedy
297 #p.plot_runs(1000,1000,0.7) # 70% greedy; 30% any_conflict

```

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

4.6 Discrete Optimization

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

```

_____cspSoft.py — Representations of Soft Constraints_____
11 from cspProblem import Variable, Constraint, CSP
12 class SoftConstraint(Constraint):
13     """A Constraint consists of
14     * scope: a tuple of variables
15     * function: a real-valued costs function that can applied to a tuple of
      values
16     * string: a string for printing the constraints. All of the strings
      must be unique.
17     for the variables
18     """
19     def __init__(self, scope, function, string=None, position=None):
20         Constraint.__init__(self, scope, function, string, position)
21
22     def value(self, assignment):
23         return self.holds(assignment)

```

```

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

```

```

32     else: return (0 if b==1 else 4 if b==2 else 3)
33 c1 = SoftConstraint([A,B],c1fun,"c1")
34 def c2fun(b,c):
35     if b==1: return (5 if c==1 else 2)
36     elif b==2: return (0 if c==1 else 4)
37     else: return (2 if c==1 else 0)
38 c2 = SoftConstraint([B,C],c2fun,"c2")
39 def c3fun(b,d):
40     if b==1: return (3 if d==1 else 0)
41     elif b==2: return 2
42     else: return (2 if d==1 else 4)
43 c3 = SoftConstraint([B,D],c3fun,"c3")
44
45 def penalty_if_same(pen):
46     "returns a function that gives a penalty of pen if the arguments are
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 65) to solve soft CSP problems.

```

cspSoft.py — (continued)
60 from display import Displayable
61 import math
62
63 class DF_branch_and_bound_opt(Displayable):
64     """returns a branch and bound searcher for a problem.
65     An optimal assignment with cost less than bound can be found by calling
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         """

```

```

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

```

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

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

Propositions and Inference

5.1 Representing Knowledge Bases

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

```
_____logicProblem.py — Representations Logics _____
11 class Clause(object):
12     """A definite clause"""
13
14     def __init__(self, head, body=[]):
15         """clause with atom head and 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 f"{self.head} <- {' & '.join(str(a) for a in
24                 self.body)}."
25         else:
26             return f"{self.head}."
```

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

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

```

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

```

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

```

logicProblem.py — (continued)
42 from display import Displayable
43
44 class KB(Displayable):
45     """A knowledge base consists of a set of clauses.
46     This also creates a dictionary to give fast access to the clauses with
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].append(c)
62         else:
63             self.atom_to_clauses[c.head] = [c]
64
65     def clauses_for_atom(self, a):
66         """returns list 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 []
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) used 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 in the intended interpretation. One of the clauses is wrong; can you see which one? We will show how to debug it.

```

logicProblem.py — (continued)
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{kb}` 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`:

```

_____logicBottomUp.py — (continued) _____
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 , unless it is needed for some other clause.

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

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

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

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

5.3 Top-down Proofs (with askables)

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

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

```

_____logicTopDown.py — Top-down Proof Procedure for Definite Clauses_____
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`:

```

_____logicTopDown.py — (continued)_____
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, which is useful if someone given an incorrect response.

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

5.4 Debugging and Explanation

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

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

```

_____logicExplain.py — Explaining Proof Procedure for Definite Clauses_____
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 |
52 | if __name__ == "__main__":
53 |     test()
54 |
55 | # try
56 | from logicProblem import elect, elect_bug
57 | # elect.max_display_level=3 # give detailed trace
58 | # prove_atom(elect, 'live_w6')
59 | # prove_atom(elect, 'lit_l1')

```

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

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

```

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

```

```

65 up          go back up proof tree to explore other parts of the proof tree
66 kb          print the knowledge base
67 quit        quit this interaction (and go back to Python)
68 help        print this text
69 """
70
71 def interact(kb):
72     going = True
73     ups = [] # stack for going up
74     proof="fail" # there is no proof to start
75     while going:
76         inp = input("logicExplain: ")
77         inps = inp.split(" ")
78         try:
79             command = inps[0]
80             if command == "quit":
81                 going = False
82             elif command == "ask":
83                 proof = prove_atom(kb, inps[1])
84                 if proof == "fail":
85                     print("fail")
86             else:
87                 print("yes")
88             elif command == "how":
89                 if proof=="fail":
90                     print("there is no proof")
91                 elif len(inps)==1:
92                     print_rule(proof)
93             else:
94                 try:
95                     ups.append(proof)
96                     proof = proof[1][int(inps[1])] #nth argument of rule
97                     print_rule(proof)
98                 except:
99                     print('In "how n", n must be a number between 0
100                        and',len(proof[1])-1,"inclusive.")
101             elif command == "up":
102                 if ups:
103                     proof = ups.pop()
104                 else:
105                     print("No rule to go up to.")
106                     print_rule(proof)
107             elif command == "kb":
108                 print(kb)
109             elif command == "help":
110                 print helptext
111             else:
112                 print("unknown command:", inp)
113                 print("use help for help")
114         except:

```

```

114         print("unknown command:", inp)
115         print("use help for help")
116
117     def print_rule(proof):
118         (head,body) = proof
119         if body == "answered":
120             print(head,"was answered yes")
121         elif body == []:
122             print(head,"is a fact")
123         else:
124             print(head,"<-")
125             for i,a in enumerate(body):
126                 print(i,":",a[0])
127
128     # try
129     # interact(elect)
130     # Which clause is wrong in elect_bug? Try:
131     # interact(elect_bug)
132     # logicExplain: ask lit_l1

```

The following shows an interaction for the knowledge base elect:

```

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

```



```

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

```

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

5.5 Assumables

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

```

_____logicAssumables.py — Definite clauses with assumables_____
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
50                         self.prove_all_ass(cl.body+ans_body[1:], assumed)
51                         ] # union of answers for each clause with
52                         head=selected
53     else:
54         # empty body
55         return [assumed] # one answer
56
57 def conflicts(self):
58     """returns a list of minimal conflicts"""
59     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 variable *ls* is referenced in the loop). For example, try to predict and then test:

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

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

```

_____logicAssumables.py — (continued)_____
69 def diagnoses(cons):
70     """cons is a list of (minimal) conflicts.

```

```

71     returns a list of diagnoses."""
72     if cons == []:
73         return [set()]
74     else:
75         return minsets([(e)|d)          # | is set union
76                        for e in cons[0]
77                        for d in diagnoses(cons[1:])])

```

Test cases:

logicAssumables.py — (continued)

```

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

```

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

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

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

5.6 Negation-as-failure

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

```

_____logicNegation.py — Propositional negation-as-failure _____
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`:

```

_____logicNegation.py — (continued) _____
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:
```

```

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

```

Test cases:

```

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

```

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

```

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

```

```
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.
- a cost for the action

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

```

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

```

stripsProblem.py — (continued)
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.

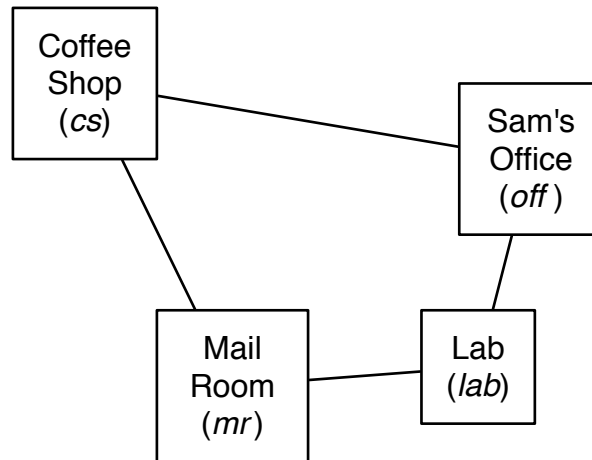
**Features to describe states***RLoc* – Rob's location*RHC* – Rob has coffee*SWC* – Sam wants coffee*MW* – Mail is waiting*RHM* – Rob has mail**Actions***mc* – move clockwise*mcc* – move counterclockwise*puc* – pickup coffee*dc* – deliver coffee*pum* – pickup mail*dm* – deliver mail

Figure 6.1: Robot Delivery Domain

```

stripsProblem.py — (continued)
53 boolean = {False, True}
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)

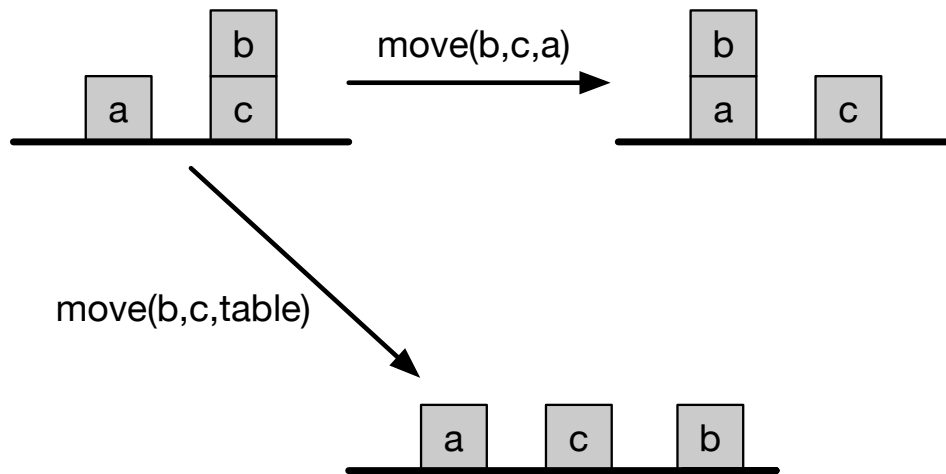


Figure 6.2: Blocks world with two actions

```

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 all the combinations of the blocks. Note that we treat moving to a block separately from moving to the table, because the blocks needs to be clear, but the table always has room for another block.

```

stripsProblem.py — (continued)
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. This example is challenging because you can't achieve one of the goals (using the minimum number of actions) and then the other; whichever one you achieve first has to be undone to achieve the second.

```

stripsProblem.py — (continued)
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'})

```

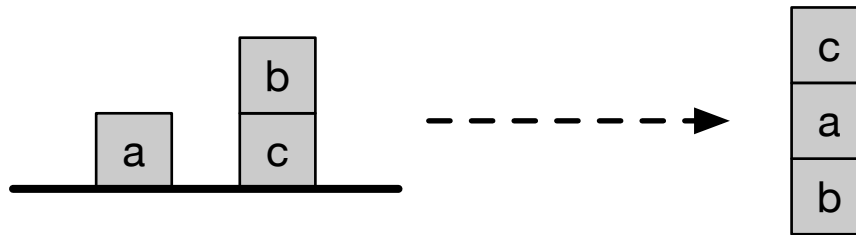


Figure 6.3: Blocks problem blocks1

```

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 include negative *on* facts? Why or why not? (Note that this may depend on the planner; write your answer with respect to particular planners.)

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

6.2 Forward Planning

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

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

```

stripsForwardPlanner.py — Forward Planner with STRIPS actions
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)

```

To define a search problem (page 41), you 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
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 import stripsProblem
87
88 # SearcherMPP(Forward_STRIPS(stripsProblem.problem1)).search() #A* with MPP
89 # DF_branch_and_bound(Forward_STRIPS(stripsProblem.problem1), 10).search()
90 #B&B
91 # To find more than one plan:
92 # s1 = SearcherMPP(Forward_STRIPS(stripsProblem.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 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 import stripsProblem
52
53 def test_forward_heuristic(thisproblem=stripsProblem.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 h1, with just h2 and with both. Explain why each one prunes or doesn't prune the search space.

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

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

Exercise 6.6 Create an admissible heuristic for the blocks world.

6.3 Regression Planning

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

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

```

stripsRegressionPlanner.py — Regression Planner with STRIPS actions
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):

```

```

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

```

stripsRegressionPlanner.py — (continued)

```

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

```

Exercise 6.7 Multiple path pruning could be used to prune more than the current node. In particular, if the current node contains more conditions than a previously visited node, it can be pruned. For example, if $\{a:\text{True}, b:\text{False}\}$ 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 will not either. Implement this more severe pruning. (Hint: This may require modifications to the searcher.)

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

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

6.3.1 Defining Heuristics for a Regression Planner

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

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

```

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

```

Exercise 6.10 Try the regression planner with a heuristic function of just 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 heuristic that is better for regression planning than `heuristic_fun` defined in Section 6.2.1.

6.4 Planning as a CSP

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

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

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

```

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

```

```

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                              f"{feat}[0]={val}"))
35                      for (feat, val) in initial_state.items()]
36
37     # goal constraints on the final state:
38     constraints += [Constraint([feat_time_var[feat][number_stages]],
39                               is_(val),
40                               f"{feat}[{number_stages}]=val"))
41                      for (feat, val) in goal.items()]
42
43     # precondition constraints:
44     constraints += [Constraint([feat_time_var[feat][t],
45                               self.action_vars[t]],
46                              if_(val, act),
47                              f"{feat}[{t}]=val if action[{t}]=act"))
48                      for act in prob_domain.actions
49                      for (feat, val) in act.preconds.items()
50                      for t in range(number_stages)]
51
52     # effect constraints:
53     constraints += [Constraint([feat_time_var[feat][t+1],
54                               self.action_vars[t]],
55                              if_(val, act),
56                              f"{feat}[{t+1}]=val if action[{t}]=act"))
57                      for act in prob_domain.actions
58                      for (feat, val) in act.effects.items()
59                      for t in range(number_stages)]
60
61     # frame constraints:
62     constraints += [Constraint([feat_time_var[feat][t],
63                               self.action_vars[t], feat_time_var[feat][t+1]],
64                              eq_if_not_in_({act for act in
65                                              prob_domain.actions
66                                              if feat in act.effects})),
67                      f"{feat}[t]={feat}[{t+1]} if act not in
68                      {set(act for act in prob_domain.actions
69                          if feat in act.effects)}"))
70                      for feat in prob_domain.feature_domain_dict
71                      for t in range(number_stages) ]
72     variables = set(self.action_vars) | {feat_time_var[feat][t]
73                                          for feat in

```

```

66         prob_domain.feature_domain_dict
67         for t in range(number_stages+1)}
68     CSP.__init__(self, "CSP_from_Strips", variables, constraints)
69
69     def extract_plan(self, soln):
70         return [soln[a] for a in self.action_vars]

```

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

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

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

```

stripsCSPPlanner.py — (continued)
72 def is_(val):
73     """returns a function that is true when it is it applied to val.
74     """
75     #return lambda x: x == val
76     def is_fun(x):
77         return x == val
78     is_fun.__name__ = f"value_is_{val}"
79     return is_fun
80
81 def if_(v1,v2):
82     """if the second argument is v2, the first argument must be v1"""
83     #return lambda x1,x2: x1==v1 if x2==v2 else True
84     def if_fun(x1,x2):
85         return x1==v1 if x2==v2 else True
86     if_fun.__name__ = f"if x2 is {v2} then x1 is {v1}"
87     return if_fun
88
89 def eq_if_not_in(actset):
90     """first and third arguments are equal if action is not in actset"""
91     # return lambda x1, a, x2: x1==x2 if a not in actset else True
92     def eq_if_not_fun(x1, a, x2):
93         return x1==x2 if a not in actset else True
94     eq_if_not_fun.__name__ = f"first and third arguments are equal if
95         action is not in {actset}"
96     return eq_if_not_fun

```

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

```

stripsCSPPlanner.py — (continued)
97 def con_plan(prob,horizon):
98     """finds a plan for problem prob given horizon.
99     """
100     csp = CSP_from_STRIPES(prob, horizon)
101     sol = Con_solver(csp).solve_one()
102     return csp.extract_plan(sol) if sol else sol

```

The following are some example queries.

```

stripsCSPPlanner.py — (continued)
104 from searchGeneric import Searcher
105 from cspConsistency import Search_with_AC_from_CSP, Con_solver
106 from stripsProblem import Planning_problem
107 import stripsProblem
108
109 # Problem 0
110 # con_plan(stripsProblem.problem0,1) # should it succeed?
111 # con_plan(stripsProblem.problem0,2) # should it succeed?
112 # con_plan(stripsProblem.problem0,3) # should it succeed?
113 # To use search to enumerate solutions
114 #searcher0a =
115     Searcher(Search_with_AC_from_CSP(CSP_from_STRIPES(stripsProblem.problem0,
116     1)))
117 #print(searcher0a.search()) # returns path to solution
118
119 ## Problem 1
120 # con_plan(stripsProblem.problem1,5) # should it succeed?
121 # con_plan(stripsProblem.problem1,4) # should it succeed?
122 ## To use search to enumerate solutions:
123 #searcher15a =
124     Searcher(Search_with_AC_from_CSP(CSP_from_STRIPES(stripsProblem.problem1,
125     5)))
126 #print(searcher15a.search()) # returns path to solution
127
128 ## Problem 2
129 #con_plan(stripsProblem.problem2, 6) # should fail??
130 #con_plan(stripsProblem.problem2, 7) # should succeed???
131
132 ## Example 6.13
133 problem3 = Planning_problem(stripsProblem.delivery_domain,
134                             {'SWC':True, 'RHC':False}, {'SWC':False})
135 #con_plan(problem3,2) # Horizon of 2
136 #con_plan(problem3,3) # Horizon of 3
137
138 problem4 = Planning_problem(stripsProblem.delivery_domain,{'SWC':True},
139                             {'SWC':False, 'MW':False, 'RHM':False})
140
141 # For the stochastic local search:
142 #from cspSLS import SLSearcher, Runtime_distribution

```

```

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

```

6.5 Partial-Order Planning

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

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

```

_____stripsPOP.py — Partial-order Planner using STRIPS representation_____
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 partial-order planner is represented as a search problem (Section 3.1) where a node consists of:

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

- *agenda*: a list of (s, a) pairs, where s is a (var, val) pair and a is an action instance. This means that variable var must have value val before a can occur.
- *causal_links*: a set of (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
64                                 other_acts)])
64         sorted_acts.append(a)
65         other_acts.remove(a)
66     return sorted_acts

```

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

```

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

```

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

```

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

```

```

101         POP_node(node.actions,consts2,new_agenda,new_cls),
102         cost=0)
103     for a0 in self.planning_problem.prob_domain.actions: #a0 is an
        action
104         if self.achieves(a0, subgoal):
105             #a0 achieves subgoal
106             new_a = Action_instance(a0)
107             self.display(2," using new action",new_a)
108             new_actions = node.actions + [new_a]
109             consts1 =
                self.add_constraint((self.start,new_a),node.constraints)
110             consts2 = self.add_constraint((new_a,act1),consts1)
111             new_agenda1 = new_agenda + [(pre,new_a) for pre in
                a0.preconds.items()]
112             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 causal link $(a0, subgoal, a1)$, the following method protects the causal link from each action in *actions*. Whenever an action deletes *subgoal*, the action needs to be before *a0* or after *a1*. This method enumerates all constraints that result from protecting the causal link from all actions.

```

stripsPOP.py — (continued)
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
157                 e
158     else:
159         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 import stripsProblem
200
201 rplanning0 = POP_search_from_STRIPS(stripsProblem.problem0)
202 rplanning1 = POP_search_from_STRIPS(stripsProblem.problem1)
203 rplanning2 = POP_search_from_STRIPS(stripsProblem.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 first chapter on machine learning 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; the code in this chapter constructs them explicitly in what is often known as **feature engineering**.
- Learning with no input features: this is the base case of many methods. What should you predict if you have no input features? This provides the base cases for many algorithms (e.g., decision tree algorithm) and baselines that more sophisticated algorithms need to beat. It also provides ways to test various predictors.
- Decision tree learning: one of the classic and simplest learning algorithms, which is the basis of many other algorithms.
- Cross validation and parameter tuning: methods to prevent overfitting.
- Linear regression and classification: other classic and simple techniques that often work well (particularly combined with feature learning or engineering).
- Boosting: combining simpler learning methods to make even better learners.

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

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

Figure 7.1: Some of the datasets used here.

7.1 Representations of Data and Predictions

The code uses the following definitions and conventions:

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

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

$f.frange$, the set of values of f seen in the dataset, represented as a list.

The $f.type$ 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]
```

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

ds.train a list of the training examples

ds.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, one_hot=False,
24                 seed=None): #12345):
25         """A dataset for learning.
26         train is a list of tuples representing the training examples
27         test is the list of tuples representing the test examples
28         if test is None, a test set is created by selecting each
29         example with probability prob_test
30         target_index is the index of the target.
31         If negative, it counts from right.
32         If target_index is larger than the number of properties,
33         there is no target (for unsupervised learning)
34         header is a list of names for the features
35         target_type is either None for automatic detection of target type
36         or one of "numeric", "boolean", "categorical"
37         one_hot is True gives a one-hot encoding of categorical features
38         seed is for random number; None gives a different test set each time
39         """
40         if seed: # given seed makes partition consistent from run-to-run
41             random.seed(seed)
42         if test is None:
43             train,test = partition_data(train, prob_test)
44         self.train = train
45         self.test = test
46
47         self.display(1,"Training set has",len(train),"examples. Number of
48             columns: ",{len(e) for e in train})
49         self.display(1,"Test set has",len(test),"examples. Number of
50             columns: ",{len(e) for e in test})
51         self.prob_test = prob_test
52         self.num_properties = len(self.train[0])
53         if target_index < 0: #allows for -1, -2, etc.
54             self.target_index = self.num_properties + target_index
55         else:
56             self.target_index = target_index
57         self.header = header
58         self.domains = [set() for i in range(self.num_properties)]

```

```

56     for example in self.train:
57         for ind,val in enumerate(example):
58             self.domains[ind].add(val)
59     self.conditions_cache = {} # cache for computed conditions
60     self.create_features(one_hot)
61     if target_type:
62         self.target.ftype = target_type
63     self.display(1,"There are",len(self.input_features),"input
        features")
64
65     def __str__(self):
66         if self.train and len(self.train)>0:
67             return ("Data: "+str(len(self.train))+ " training examples, "
68                     +str(len(self.test))+ " test examples, "
69                     +str(len(self.train[0]))+" features.")
70         else:
71             return ("Data: "+str(len(self.train))+ " training examples, "
72                     +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)
74     def create_features(self, one_hot=False):
75         """create the set of features.
76         if one_hot==True then make categorical features into booleans for
77         each value
78         """
79         self.target = None
80         self.input_features = []
81         for i in range(self.num_properties):
82             frange = list(self.domains[i])
83             ftype = self.infer_type(frange)
84             if one_hot and ftype == "categorical" and i !=
                self.target_index:
85                 for val in frange:
86                     def feat(e,index=i,val=val):
87                         return e[index]==val
88                     if self.header:
89                         feat.__doc__ = self.header[i]+"="+val
90                     else:
91                         feat.__doc__ = f"e[{i}]={val}"
92                     feat.frange = boolean
93                     feat.type = "boolean"
94                     self.input_features.append(feat)
95             else:
96                 def feat(e,index=i):
97                     return e[index]
98                 if self.header:

```

```

98         feat.__doc__ = self.header[i]
99     else:
100         feat.__doc__ = "e["+str(i)+"]"
101         feat.frange = frange
102         feat.ftype = ftype
103         if i == self.target_index:
104             self.target = feat
105     else:
106         self.input_features.append(feat)

```

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

```

learnProblem.py — (continued)
108 def infer_type(self, domain):
109     """Infers the type of a feature with domain
110     """
111     if all(v in {True, False} for v in domain) or all(v in {0, 1} for v
112             in domain):
113         return "boolean"
114     if all(isinstance(v, (float, int)) for v in domain):
115         return "numeric"
116     else:
117         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, the following code constructs Boolean conditions from the attributes.

There are 3 cases:

- When the range only has two values, one is designated to be the “true” value.
- When the values are all numeric, assume they are ordered (as opposed to just being some classes that happen to be labelled with numbers) and construct Boolean features for splits of the data. That is, the feature is $e[ind] < cut$ for some value cut . The number of cut values is less than or equal to max_num_cuts .
- When the values are not all numeric, it creates an indicator function for each value. An indicator function for a value returns true when that value is given and false otherwise. Note that we can’t create an indicator function for values that appear in the test set but not in the training 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 create only Boolean features for categorical input features, and not to make cuts for numerical values.

```

118 def conditions(self, max_num_cuts=8, categorical_only = False):
119     """returns a list of boolean conditions from the input features
120     max_num_cuts is the maximum number of cuts for numeric features
121     categorical_only is true if only categorical features are made
122         binary
123     """
124     if (max_num_cuts, categorical_only) in self.conditions_cache:
125         return self.conditions_cache[(max_num_cuts, categorical_only)]
126     conds = []
127     for ind, frange in enumerate(self.domains):
128         if ind != self.target_index and len(frange)>1:
129             if len(frange) == 2:
130                 # two values, the feature is equality to one of them.
131                 true_val = list(frange)[1] # choose one as true
132                 def feat(e, i=ind, tv=true_val):
133                     return e[i]==tv
134                 if self.header:
135                     feat.__doc__ = f"{self.header[ind]}=={true_val}"
136                 else:
137                     feat.__doc__ = f"e[{ind]}=={true_val}"
138                 feat.frange = boolean
139                 feat.ftype = "boolean"
140                 conds.append(feat)
141             elif all(isinstance(val, (int, float)) for val in frange):
142                 if categorical_only: # numeric, don't make cuts
143                     def feat(e, i=ind):
144                         return e[i]
145                     feat.__doc__ = f"e[{ind}]"
146                     conds.append(feat)
147                 else:
148                     # all numeric, create cuts of the data
149                     sorted_frange = sorted(frange)
150                     num_cuts = min(max_num_cuts, len(frange))
151                     cut_positions = [len(frange)*i//num_cuts for i in
152                                     range(1, num_cuts)]
153                     for cut in cut_positions:
154                         cutat = sorted_frange[cut]
155                         def feat(e, ind_=ind, cutat=cutat):
156                             return e[ind_] < cutat
157                         if self.header:
158                             feat.__doc__ = self.header[ind]+"<" + str(cutat)
159                         else:
160                             feat.__doc__ = "e[" + str(ind) + "]" + "<" + str(cutat)

```

```

160         feat.frange = boolean
161         feat.ftype = "boolean"
162         conds.append(feat)
163     else:
164         # create an indicator function for every value
165         for val in frange:
166             def feat(e, ind_=ind, val_=val):
167                 return e[ind_] == val_
168             if self.header:
169                 feat.__doc__ = self.header[ind]+"==" +str(val)
170             else:
171                 feat.__doc__ = "e["+str(ind)+"]==" +str(val)
172             feat.frange = boolean
173             feat.ftype = "boolean"
174             conds.append(feat)
175     self.conditions_cache[(max_num_cuts, categorical_only)] = conds
176     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. Three evaluation criteria are implemented, 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)
178 def evaluate_dataset(self, data, predictor, error_measure):
179     """Evaluates predictor on data according to the error_measure
180     predictor is a function that takes an example and returns a
181     prediction for the target features.
182     error_measure(prediction,actual) -> non-negative real
183     """
184     if data:
185         try:
186             value = statistics.mean(error_measure(predictor(e),
187                                     self.target(e))
188                                     for e in data)
189         except ValueError: # if error_measure gives an error
190             return float("inf") # infinity
191         return value
192     else:
193         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)
194 class Evaluate(object):
195     """A container for the evaluation measures"""
196
197     def squared_loss(prediction, actual):
198         "squared loss "
199         if isinstance(prediction, (list,dict)):
200             return (1-prediction[actual])**2 # the correct value is 1
201         else:
202             return (prediction-actual)**2
203
204     def absolute_loss(prediction, actual):
205         "absolute loss "
206         if isinstance(prediction, (list,dict)):
207             return abs(1-prediction[actual]) # the correct value is 1
208         else:
209             return abs(prediction-actual)
210
211     def log_loss(prediction, actual):
212         "log loss (bits)"
213         try:
214             if isinstance(prediction, (list,dict)):
215                 return -math.log2(prediction[actual])
216             else:
217                 return -math.log2(prediction) if actual==1 else
218                     -math.log2(1-prediction)
219         except ValueError:

```

```

219         return float("inf") # infinity
220
221     def accuracy(prediction, actual):
222         "accuracy"
223         if isinstance(prediction, dict):
224             prev_val = prediction[actual]
225             return 1 if all(prev_val >= v for v in prediction.values())
226                 else 0
227         if isinstance(prediction, list):
228             prev_val = prediction[actual]
229             return 1 if all(prev_val >= v for v in prediction) else 0
230         else:
231             return 1 if abs(actual-prediction) <= 0.5 else 0
232
233     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 it may not know, as data may just be a generator of the data (e.g., when reading the data from a file).]

```

learnProblem.py — (continued)
234 def partition_data(data, prob_test=0.30):
235     """partitions the data into a training set and a test set, where
236     prob_test is the probability of each example being in the test set.
237     """
238     train = []
239     test = []
240     for example in data:
241         if random.random() < prob_test:
242             test.append(example)
243         else:
244             train.append(example)
245     return train, test

```

7.1.4 Importing Data From File

A dataset is typically loaded from a file. The default here is that it loaded from a CSV (comma separated values) file, although the separator can be changed. This assumes that all lines that contain the separator are valid data (so it only includes those data items that contain more than one element). This allows for

blank lines and comment lines that do not contain the separator. However, it means that this method is not suitable for cases where there is only one feature.

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

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)
247 class Data_from_file(Data_set):
248     def __init__(self, file_name, separator=',', num_train=None,
        prob_test=0.3,
249         has_header=False, target_index=0, one_hot=False,
250         categorical=[], target_type= None, include_only=None,
        seed=None): #seed=12345):
251         """create a dataset from a file
252         separator is the character that separates the attributes
253         num_train is a number specifying the first num_train tuples are
        training, or None
254         prob_test is the probability an example should in the test set (if
        num_train is None)
255         has_header is True if the first line of file is a header
256         target_index specifies which feature is the target
257         one_hot specifies whether categorical features should be encoded as
        one_hot.
258         categorical is a set (or list) of features that should be treated
        as categorical
259         target_type is either None for automatic detection of target type
260         or one of "numeric", "boolean", "categorical"
261         include_only is a list or set of indexes of columns to include
262         """
263         with open(file_name,'r',newline='') as csvfile:
264             self.display(1,"Loading",file_name)
265             # data_all = csv.reader(csvfile,delimiter=separator) # for more
        complicated CSV files
266             data_all = (line.strip().split(separator) for line in csvfile)
267             if include_only is not None:
268                 data_all = ([v for (i,v) in enumerate(line) if i in
        include_only]
269                     for line in data_all)
270             if has_header:
271                 header = next(data_all)
272             else:
273                 header = None

```



```

274         data_tuples = (interpret_elements(d) for d in data_all if
275                         len(d)>1)
276         if num_train is not None:
277             # training set is divided into training then text examples
278             # the file is only read once, and the data is placed in
279             # appropriate list
280             train = []
281             for i in range(num_train): # will give an error if
282                 # insufficient examples
283                 train.append(next(data_tuples))
284             test = list(data_tuples)
285             Data_set.__init__(self, train, test=test,
286                             target_index=target_index, header=header)
287         else: # randomly assign training and test examples
288             Data_set.__init__(self, data_tuples, test=None,
289                             prob_test=prob_test,
290                             target_index=target_index, header=header,
291                             seed=seed, target_type=target_type,
292                             one_hot=one_hot)

```

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

```

learnProblem.py — (continued)
287 class Data_from_files(Data_set):
288     def __init__(self, train_file_name, test_file_name, separator=',',
289                 has_header=False, target_index=0, one_hot=False,
290                 categorical=[], target_type= None, include_only=None):
291         """create a dataset from separate training and file
292         separator is the character that separates the attributes
293         num_train is a number specifying the first num_train tuples are
294             training, or None
295         prob_test is the probability an example should in the test set (if
296             num_train is None)
297         has_header is True if the first line of file is a header
298         target_index specifies which feature is the target
299         one_hot specifies whether categorical features should be encoded as
300             one-hot
301         categorical is a set (or list) of features that should be treated
302             as categorical
303         target_type is either None for automatic detection of target type
304             or one of "numeric", "boolean", "categorical"
305         include_only is a list or set of indexes of columns to include
306         """
307         with open(train_file_name, 'r', newline='') as train_file:
308             with open(test_file_name, 'r', newline='') as test_file:
309                 # data_all = csv.reader(csvfile, delimiter=separator) # for more
310                 # complicated CSV files
311                 train_data = (line.strip().split(separator) for line in
312                             train_file)

```

```

307         test_data = (line.strip().split(separator) for line in
308                       test_file)
309         if include_only is not None:
310             train_data = ([v for (i,v) in enumerate(line) if i in
311                             include_only]
312                            for line in train_data)
313             test_data = ([v for (i,v) in enumerate(line) if i in
314                             include_only]
315                           for line in test_data)
316         if has_header: # this assumes the training file has a header
317             and the test file doesn't
318             header = next(train_data)
319         else:
320             header = None
321         train_tuples = [interpret_elements(d) for d in train_data if
322                         len(d)>1]
323         test_tuples = [interpret_elements(d) for d in test_data if
324                        len(d)>1]
325         Data_set.__init__(self,train_tuples, test_tuples,
326                           target_index=target_index, header=header,
327                           one_hot=one_hot)

```

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

```

learnProblem.py — (continued)
322 def interpret_elements(str_list):
323     """make the elements of string list str_list numeric if possible.
324     Otherwise remove initial and trailing spaces.
325     """
326     res = []
327     for e in str_list:
328         try:
329             res.append(int(e))
330         except ValueError:
331             try:
332                 res.append(float(e))
333             except ValueError:
334                 se = e.strip()
335                 if se in ["True", "true", "TRUE"]:
336                     res.append(True)
337                 elif se in ["False", "false", "FALSE"]:
338                     res.append(False)
339                 else:
340                     res.append(e.strip())
341     return res

```

7.1.5 Augmented Features

Sometimes we want to augment the features with new features computed from the old features (e.g., the product of features). The following code creates a new dataset from an old dataset but with new features. Note that special cases of these are **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)
343 class Data_set_augmented(Data_set):
344     def __init__(self, dataset, unary_functions=[], binary_functions=[],
345               include_orig=True):
346         """creates a dataset like dataset but with new features
347         unary_function is a list of unary feature constructors
348         binary_functions is a list of binary feature combiners.
349         include_orig specifies whether the original features should be
350         included
351         """
352         self.orig_dataset = dataset
353         self.unary_functions = unary_functions
354         self.binary_functions = binary_functions
355         self.include_orig = include_orig
356         self.target = dataset.target
357         Data_set.__init__(self, dataset.train, test=dataset.test,
358                           target_index = dataset.target_index)
359
360     def create_features(self, one_hot=False):
361         """create the set of features.
362         one_hot is ignored, but could be implemented as in
363         Data_set.create_features
364         """
365         if self.include_orig:
366             self.input_features = self.orig_dataset.input_features.copy()
367         else:
368             self.input_features = []
369         for u in self.unary_functions:
370             for f in self.orig_dataset.input_features:
371                 self.input_features.append(u(f))
372         for b in self.binary_functions:
373             for f1 in self.orig_dataset.input_features:
374                 for f2 in self.orig_dataset.input_features:
375                     if f1 != f2:
376                         self.input_features.append(b(f1, f2))

```

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

```

learnProblem.py — (continued)
375 def square(f):
376     """a unary feature constructor to construct the square of a feature
377     """
378     def sq(e):
379         return f(e)**2
380     sq.__doc__ = f.__doc__+"**2"
381     return sq
382
383 def power_feat(n):
384     """given n returns a unary feature constructor to construct the nth
385     power of a feature.
386     e.g., power_feat(2) is the same as square, defined above
387     """
388     def fn(f,n=n):
389         def pow(e,n=n):
390             return f(e)**n
391         pow.__doc__ = f.__doc__+"**"+str(n)
392         return pow
393     return fn
394
395 def prod_feat(f1,f2):
396     """a new feature that is the product of features f1 and f2
397     """
398     def feat(e):
399         return f1(e)*f2(e)
400     feat.__doc__ = f1.__doc__+"*"+f2.__doc__
401     return feat
402
403 def eq_feat(f1,f2):
404     """a new feature that is 1 if f1 and f2 give same value
405     """
406     def feat(e):
407         return 1 if f1(e)==f2(e) else 0
408     feat.__doc__ = f1.__doc__+"==" +f2.__doc__
409     return feat
410
411 def neq_feat(f1,f2):
412     """a new feature that is 1 if f1 and f2 give different values
413     """
414     def feat(e):
415         return 1 if f1(e)!=f2(e) else 0
416     feat.__doc__ = f1.__doc__+"!=" +f2.__doc__
417     return feat

```

Example:

```
learnProblem.py — (continued)
```

```

418 # from learnProblem import Data_set_augmented, prod_feat
419 # data = Data_from_file('data/holiday.csv', has_header=True, num_train=19,
      target_index=-1)
420 # data = Data_from_file('data/iris.data', prob_test=1/3, target_index=-1)
421 ## Data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0)
422 # dataplus = Data_set_augmented(data, [], [prod_feat])
423 # dataplus = Data_set_augmented(data, [], [prod_feat, neq_feat])

```

Exercise 7.3 For symmetric properties, such as product, we don't need both $f_1 * f_2$ as well as $f_2 * f_1$ as extra properties. Allow the user to be able to declare feature constructors as symmetric (by associating a Boolean feature with them). Change *construct_features* so that it does not create both versions for symmetric combiners.

7.2 Generic Learner Interface

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

```

learnProblem.py — (continued)
424 from display import Displayable
425
426 class Learner(Displayable):
427     def __init__(self, dataset):
428         raise NotImplementedError("Learner.__init__") # abstract method
429
430     def learn(self):
431         """returns a predictor, a function from a tuple to a value for the
            target feature
432         """
433         raise NotImplementedError("learn") # abstract method

```

7.3 Learning With No Input Features

If you need make the same prediction for each example, what prediction should you make? This can be used as a naive baseline; if a more sophisticated method does not do better than this, it is not useful. This also provides the base case for some methods, such as decision-tree learning.

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

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

- a point prediction, where 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 defined, there is no point in predicting a value greater than 1 or less than zero (but that doesn't mean you 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.

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 the middle values when there are an even number of examples, whereas `median` gives the average of them (and so `cmedian` is applicable for ordinals that cannot be considered cardinal values). Similarly, `cmode` picks one of the values when more than one value has the maximum number of elements.

```

learnNoInputs.py — Learning ignoring all input features
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"
```

```

33         return {k:min(max(v,bound),1-bound) for (k,v) in
34                    Predict.empirical(data, domain).items()}
35
36 def laplace(data, domain=[0,1]):
37     "Laplace      " # for categorical data
38     return Predict.empirical(data, domain, icount=1)
39
40 def cmode(data, domain=[0,1]):
41     "mode         " # for categorical data
42     md = statistics.mode(data)
43     return {v: 1 if v==md else 0 for v in domain}
44
45 def cmedian(data, domain=[0,1]):
46     "median       " # for categorical data
47     md = statistics.median_low(data) # always return one of the values
48     return {v: 1 if v==md else 0 for v in domain}
49
50 ### The following return a single prediction (for regression). domain
51     is ignored.
52
53 def mean(data, domain=[0,1]):
54     "mean         "
55     # returns a real number
56     return statistics.mean(data)
57
58 def rmean(data, domain=[0,1], mean0=0, pseudo_count=1):
59     "regularized mean"
60     # returns a real number.
61     # mean0 is the mean to be used for 0 data points
62     # With mean0=0.5, pseudo_count=2, same as laplace for [0,1] data
63     # this works for enumerations as well as lists
64     sum = mean0 * pseudo_count
65     count = pseudo_count
66     for e in data:
67         sum += e
68         count += 1
69     return sum/count
70
71 def mode(data, domain=[0,1]):
72     "mode         "
73     return statistics.mode(data)
74
75 def median(data, domain=[0,1]):
76     "median       "
77     return statistics.median(data)
78
79 all = [empirical, mean, rmean, bounded_empirical, laplace, cmode, mode,
        median, cmedian]

```

The following suggests appropriate predictions as a function of the

```

target type
80 select = {"boolean": [empirical, bounded_empirical, laplace, cmode,
    cmedian],
81          "categorical": [empirical, bounded_empirical, laplace, cmode,
    cmedian],
82          "numeric": [mean, rmean, mode, median]}

```

7.3.1 Evaluation

To evaluate a point prediction, let's first generate some possible values, 0 and 1 for the target feature. Given the ground truth *prob*, a number in the range [0,1], the following code generates some training and test data where *prob* is the probability of each example being 1. To generate a 1 with probability *prob*, it generates a random number in range [0,1] and return 1 if that number is less than *prob*. A prediction is computed by applying the predictor to the training data, which is evaluated on the test set. This is repeated *num_samples* times.

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

```

learnNoInputs.py — (continued)
84 def test_no_inputs(error_measures = Evaluate.all_criteria,
    num_samples=10000,
85                    test_size=10, training_sizes=
    [1,2,3,4,5,10,20,100,1000]):
86     for train_size in training_sizes:
87         results = {predictor: {error_measure: 0 for error_measure in
    error_measures}
88                    for predictor in Predict.all}
89     for sample in range(num_samples):
90         prob = random.random()
91         training = [1 if random.random()<prob else 0 for i in
    range(train_size)]
92         test = [1 if random.random()<prob else 0 for i in
    range(test_size)]
93         for predictor in Predict.all:
94             prediction = predictor(training)
95             for error_measure in error_measures:
96                 results[predictor][error_measure] += sum(
    error_measure(prediction,actual)
97                                                         for actual in
    test) /
    test_size
98     print(f"For training size {train_size}:")
99     print("  Predictor\t", "\t".join(error_measure.__doc__ for
100                                     error_measure in
    error_measures), sep="\t")
101     for predictor in Predict.all:
102         print(f"  {predictor.__doc__}",
103               "\t".join("{:.7f}".format(results[predictor][error_measure]/num_samples)

```



```

104 |                                     for error_measure in
105 |                                         error_measures), sep="\t")
106 | if __name__ == "__main__":
107 |     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                     leaf_prediction=Predict.empirical, # what to use for value
20                     train=None, # used for cross validation
21                     max_num_cuts=8, # maximum number of conditions to split a
22                     numeric feature into

```

```

22         gamma=1e-7, # minimum improvement needed to expand a node
23         min_child_weight=10):
24     self.dataset = dataset
25     self.target = dataset.target
26     self.split_to_optimize = split_to_optimize
27     self.leaf_prediction = leaf_prediction
28     self.max_num_cuts = max_num_cuts
29     self.gamma = gamma
30     self.min_child_weight = min_child_weight
31     if train is None:
32         self.train = self.dataset.train
33     else:
34         self.train = train
35
36     def learn(self, max_num_cuts=8):
37         """learn a decision tree"""
38         return self.learn_tree(self.dataset.conditions(self.max_num_cuts),
                                self.train)

```

The main recursive algorithm, takes in a set of input features and a set of training data. It first decides whether to split. If it doesn't split, it makes a point prediction, ignoring the input features.

It only splits if the best split increases the error by at least *gamma*. This implies it does not split when:

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

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

```

learnDT.py — (continued)
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)

```

```

52         self.display(2,f"leaf prediction for {len(data_subset)}
           examples is {prediction}")
53     def leaf_fun(e):
54         return prediction
55     leaf_fun.__doc__ = str(prediction)
56     leaf_fun.num_leaves = 1
57     return leaf_fun
58 else: # a split succeeded
59     false_examples, true_examples = partn
60     rem_features = [fe for fe in conditions if fe != split]
61     self.display(2,"Splitting on",split.__doc__,"with examples
           split",
62                 len(true_examples),":",len(false_examples))
63     true_tree = self.learn_tree(rem_features,true_examples)
64     false_tree = self.learn_tree(rem_features,false_examples)
65     def fun(e):
66         if split(e):
67             return true_tree(e)
68         else:
69             return false_tree(e)
70     #fun = lambda e: true_tree(e) if split(e) else false_tree(e)
71     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
88     not None
89     """
90     best_feat = None # best feature
91     # best_error = float("inf") # infinity - more than any error
92     best_error = self.sum_losses(data_subset) - self.gamma
93     self.display(3," no split has
94                 error=",best_error,"with",len(conditions),"conditions")
95     best_partition = None
96     for feat in conditions:
97         false_examples, true_examples = partition(data_subset,feat)
98         if

```

```

100         min(len(false_examples), len(true_examples)) >= self.min_child_weight:
101         err = (self.sum_losses(false_examples)
102               + self.sum_losses(true_examples))
103         self.display(3, " split on", feat.__doc__, "has error=", err,
104                     "splits
105                     into", len(true_examples), ":", len(false_examples), "gamma=", self.gamma)
106         if err < best_error:
107             best_feat = feat
108             best_error = err
109             best_partition = false_examples, true_examples
110         self.display(2, "best split is on", best_feat.__doc__,
111                     "with err=", best_error)
112         return best_feat, best_partition
113
114     def sum_losses(self, data_subset):
115         """returns sum of losses for dataset (with no more splits)
116         There a single prediction for all leaves using leaf_prediction
117         It is evaluated using split_to_optimize
118         """
119         prediction = self.leaf_value(data_subset, self.target.frange)
120         error = sum(self.split_to_optimize(prediction, self.target(e))
121                    for e in data_subset)
122         return error
123
124     def partition(data_subset, feature):
125         """partitions the data_subset by the feature"""
126         true_examples = []
127         false_examples = []
128         for example in data_subset:
129             if feature(example):
130                 true_examples.append(example)
131             else:
132                 false_examples.append(example)
133         return false_examples, true_examples

```

Test cases:

```

131 from learnDT.py — (continued)
132
133 from learnProblem import Data_set, Data_from_file
134
135 def testDT(data, print_tree=True, selections = None, **tree_args):
136     """Prints errors and the trees for various evaluation criteria and ways
137     to select leaves.
138     """
139     if selections == None: # use selections suitable for target type
140         selections = Predict.select[data.target.ftype]
141     evaluation_criteria = Evaluate.all_criteria
142     print("Split Choice", "Leaf Choice\t", "#leaves", '\t'.join(ecrit.__doc__
143                                     for ecrit in
144                                     evaluation_criteria), sep="\t")
145     for crit in evaluation_criteria:

```

```

142         for leaf in selections:
143             tree = DT_learner(data, split_to_optimize=crit,
                               leaf_prediction=leaf,
144                               **tree_args).learn()
145             print(crit.__doc__, leaf.__doc__, tree.num_leaves,
146                   "\t".join("{:.7f}".format(data.evaluate_dataset(data.test,
                               tree, ecrit))
                               for ecrit in evaluation_criteria), sep="\t")
147
148         if print_tree:
149             print(tree.__doc__)
150
151 #DT_learner.max_display_level = 4
152 if __name__ == "__main__":
153     # Choose one of the data files
154     #data=Data_from_file('data/SPECT.csv', target_index=0);
155     #print("SPECT.csv")
156     #data=Data_from_file('data/iris.data', target_index=-1);
157     #print("iris.data")
158     data = Data_from_file('data/carbool.csv', target_index=-1, seed=123)
159     #data = Data_from_file('data/mail_reading.csv', target_index=-1);
160     #print("mail_reading.csv")
161     #data = Data_from_file('data/holiday.csv', has_header=True,
162                           num_train=19, target_index=-1); print("holiday.csv")
163     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 the minimum number of examples in decision-tree search, both for the validation set and for the test set. The error on the validation set can be used to tune the parameter — choose the value of the parameter that minimizes the error. The error on the test set cannot be used to tune the parameters; if it were to be used this way it could not be used to test how well the method works on unseen examples.

```

learnCrossValidation.py — (continued)
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()

```

```

59 plt.xscale(xscale) # change between log and linear scale
60 plt.xlabel("min_child_weight")
61 plt.ylabel("average "+criterion.__doc__)
62 folded_data = K_fold_dataset(data, num_folds)
63 if maxx == None:
64     maxx = len(data.train)//2+1
65 errors = [] # validation errors
66 terrors = [] # test set errors
67 for mcw in range(1,maxx):
68     verrors.append(folded_data.validation_error(DT_learner,criterion,leaf_prediction=leaf_prediction,
69                                                  min_child_weight=mcw))
70     tree = DT_learner(data, criterion, leaf_prediction=leaf_prediction,
71                       min_child_weight=mcw).learn()
72     terrors.append(data.evaluate_dataset(data.test,tree,criterion))
73 plt.plot(range(1,maxx), verrors, ls='-',color='k',
74          label="validation for "+criterion.__doc__)
75 plt.plot(range(1,maxx), terrors, ls='--',color='k',
76          label="test set for "+criterion.__doc__)
77 plt.legend()
78 plt.draw()
79 # The following produces the graphs of Figure 7.18 of Poole and Mackworth
80 # [2023]
81 # data = Data_from_file('data/SPECT.csv',target_index=0, seed=123)
82 # plot_error(data, criterion=Evaluate.log_loss,
83 #            leaf_prediction=Predict.laplace)
84 #also try:
85 # plot_error(data)
86 # 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.

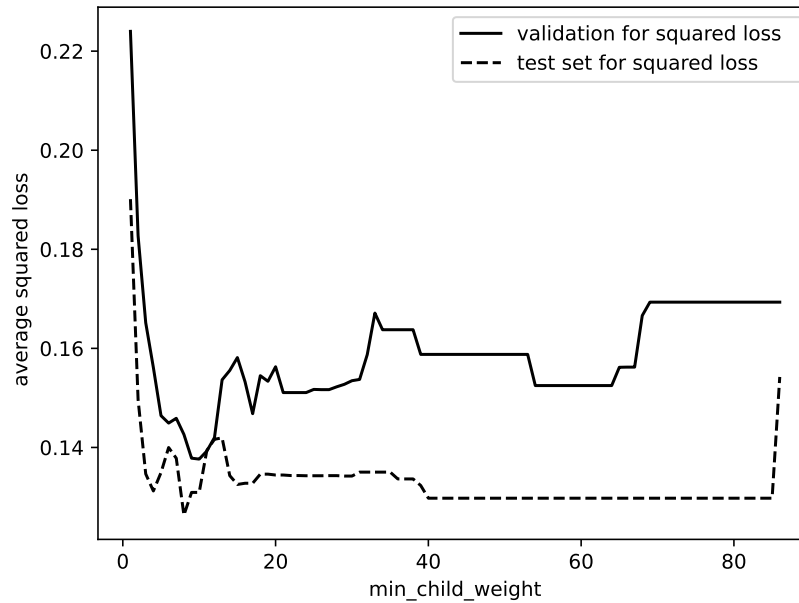


Figure 7.2: plot_error for SPECT dataset

```

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
        below
38     self.weights = {feat:random.uniform(-max_init,max_init)
39                     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
```

$\text{sigmoid}(x)$ 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)
```

$\text{softmax}([x_0, x_2, \dots])$ returns $[v_0, v_2, \dots]$ where

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

_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', has_header=True,
142             num_train=19, 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 = []

```

```

149     test_errors = []
150     for i in range(1,num_steps+1,step):
151         test_errors.append(data.evaluate_dataset(data.test,
152             learner.predictor, criterion))
153         train_errors.append(data.evaluate_dataset(data.train,
154             learner.predictor, criterion))
155         learner.display(2, "Train error:",train_errors[-1],
156             "Test error:",test_errors[-1])
157         learner.learn(num_iter=step)
158     plt.plot(range(1,num_steps+1,step),train_errors,ls='-',label=legend_label+"training")
159     plt.plot(range(1,num_steps+1,step),test_errors,ls='--',label=legend_label+"test")
160     plt.legend()
161     plt.draw()
162     learner.display(1, "Train error:",train_errors[-1],
163         "Test error:",test_errors[-1])
164
165 if __name__ == "__main__":
166     test()
167
168 # This generates the figure
169 # from learnProblem import Data_set_augmented, prod_feat
170 # data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0,
171     seed=123)
172 # dataplus = Data_set_augmented(data, [], [prod_feat])
173 # plot_steps(data=data, num_steps=1000)
174 # 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 In Figure 7.3, the log loss is very unstable when there are over 20 steps. Hypothesize why this occurs. [Hint: when does gradient descent become instable?] Test your hypothesis by running with different hyperparameters.

Exercise 7.12 The squashed learner only makes predictions in the range (0,1). If the output values are {1,2,3,4} there is no use predicting less than 1 or greater than 4. Change the squashed learner so that it can learn values in the range (1,4). Test it on the file 'data/car.csv'.

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

```

learnLinear.py — (continued)
172 def arange(start,stop,step):
173     """returns enumeration of values in the range [start,stop) separated by
174         step.
175         like the built-in range(start,stop,step) but allows for integers and
176         floats.
177         Note that rounding errors are expected with real numbers. (or use
178         numpy.arange)

```

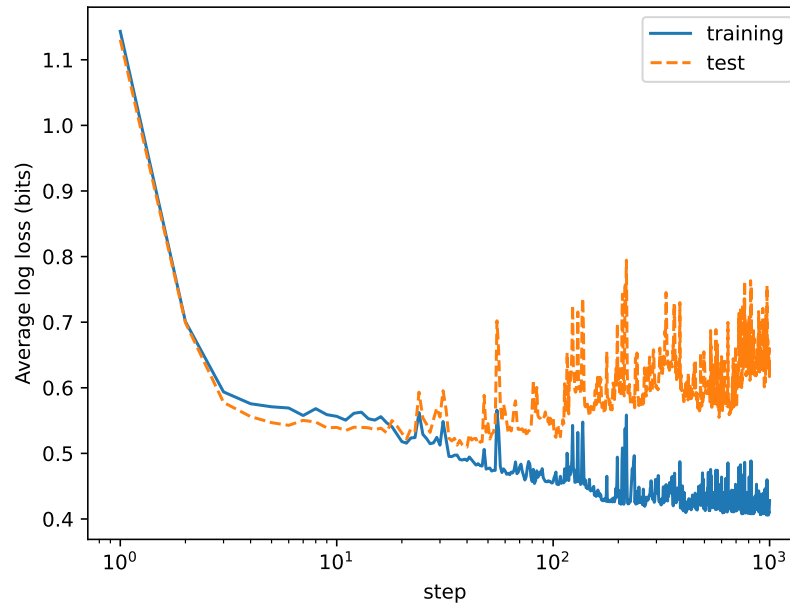


Figure 7.3: plot_steps for SPECT dataset

```

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,

```

```

    Evaluate.squared_loss))
200 plt.plot([e[0] for e in data.train],[e[-1] for e in
    data.train],"bo",label="data")
201 plt.plot(list(arange(minx,maxx,step_size)),[learner.predictor([x])
202         for x in
    arange(minx,maxx,step_size)],
203         label=label)
204 plt.legend()
205 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, one_hot=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)

```

Exercise 7.13 For each of the polynomial functions learned: What is the prediction as x gets larger ($x \rightarrow \infty$). What is the prediction as x gets more negative ($x \rightarrow -\infty$).

7.7 Boosting

The following code implements functional gradient boosting for regression.

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

```



```

39
40     def conditions(self, *args, colsample_bytree=0.5, **nargs):
41         conds = self.base_dataset.conditions(*args, **nargs)
42         return random.sample(conds, int(colsample_bytree*len(conds)))

```

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

```

----- learnBoosting.py --- (continued) -----
44 class Boosting_learner(Learner):
45     def __init__(self, dataset, base_learner_class, subsample=0.8):
46         self.dataset = dataset
47         self.base_learner_class = base_learner_class
48         self.subsample = subsample
49         mean = sum(self.dataset.target(e)
50                   for e in self.dataset.train)/len(self.dataset.train)
51         self.predictor = lambda e:mean # function that returns mean for
            each example
52         self.predictor.__doc__ = "lambda e:"+str(mean)
53         self.offsets = [self.predictor] # list of base learners
54         self.predictors = [self.predictor] # list of predictors
55         self.errors = [data.evaluate_dataset(data.test, self.predictor,
            Evaluate.squared_loss)]
56         self.display(1,"Predict test set mean squared loss=",
            self.errors[0] )
57
58
59     def learn(self, num_ensembles=10):
60         """adds num_ensemble learners to the ensemble.
61         returns a new predictor.
62         """
63         for i in range(num_ensembles):
64             train_subset = Boosted_dataset(self.dataset, self.predictor,
                subsample=self.subsample)
65             learner = self.base_learner_class(train_subset)
66             new_offset = learner.learn()
67             self.offsets.append(new_offset)
68             def new_pred(e, old_pred=self.predictor, off=new_offset):
69                 return old_pred(e)+off(e)
70             self.predictor = new_pred
71             self.predictors.append(new_pred)
72             self.errors.append(data.evaluate_dataset(data.test,
                self.predictor, Evaluate.squared_loss))
73             self.display(1,f"Iteration {len(self.offsets)-1},treesize =
                {new_offset.num_leaves}. mean squared
                loss={self.errors[-1]}")
74         return self.predictor

```

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

```

learnBoosting.py — (continued)
76 # Testing
77
78 from learnDT import DT_learner
79 from learnProblem import Data_set, Data_from_file
80
81 def sp_DT_learner(split_to_optimize=Evaluate.squared_loss,
82                  leaf_prediction=Predict.mean,**nargs):
83     """Creates a learner with different default arguments replaced by
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', has_header=True, num_train=19,
98                      target_index=-1)
99 #learner10 = Boosting_learner(data,
100                             sp_DT_learner(split_to_optimize=Evaluate.squared_loss,
101                                             leaf_prediction=Predict.mean, min_child_weight=10))
102 #learner7 = Boosting_learner(data, sp_DT_learner(0.7))
103 #learner5 = Boosting_learner(data, sp_DT_learner(0.5))
104 #predictor9 =learner9.learn(10)
105 #for i in learner9.offsets: print(i.__doc__)
106 import matplotlib.pyplot as plt
107
108 def plot_boosting_trees(data, steps=10, mcws=[30,20,20,10], gammas=
109     [100,200,300,500]):
110     # to reduce clutter uncomment one of following two lines
111     #mcws=[10]
112     #gammas=[200]
113     learners = [(mcw, gamma, Boosting_learner(data,
114         sp_DT_learner(min_child_weight=mcw, gamma=gamma)))
115                 for gamma in gammas for mcw in mcws
116                 ]
117
118     plt.ion()
119     plt.xscale('linear') # change between log and linear scale
120     plt.xlabel("number of trees")
121     plt.ylabel("mean squared loss")
122     markers = (m+c for c in ['k','g','r','b','m','c','y'] for m in
123               ['-','-.-','-',':'])
124     for (mcw,gamma,learner) in learners:
125         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,mcws=[20], gammas= [100,200,300,500])
123 # plot_boosting_trees(data,mcws=[30,20,20,10], gammas= [100])

```

Exercise 7.14 For a particular dataset, suggest good values for `min_child_weight` and `gamma`. How stable are these to different random choices that are made (e.g., in the training-test split)? Try to explain why these are good settings.

7.7.1 Gradient Tree Boosting

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

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

```

learnBoosting.py — (continued)
125 class GTB_learner(DT_learner):
126     def __init__(self, dataset, number_trees, lambda_reg=1, gamma=0,
127                 **dtargs):
128         DT_learner.__init__(self, dataset,
129                             split_to_optimize=Evaluate.log_loss, **dtargs)
130         self.number_trees = number_trees
131         self.lambda_reg = lambda_reg
132         self.gamma = gamma
133         self.trees = []
134
135     def learn(self):
136         for i in range(self.number_trees):
137             tree =
138                 self.learn_tree(self.dataset.conditions(self.max_num_cuts),
139                                 self.train)
140             self.trees.append(tree)
141             self.display(1, f"Iteration {i} treesize = {tree.num_leaves}
142                           train logloss={
143                               self.dataset.evaluate_dataset(self.dataset.train,
144                                                               self.gtb_predictor, Evaluate.log_loss)
145                           } test logloss={

```

```

140         self.dataset.evaluate_dataset(self.dataset.test,
141                                     self.gtb_predictor, Evaluate.log_loss)"""
142     return self.gtb_predictor
143
144     def gtb_predictor(self, example, extra=0):
145         """prediction for example,
146         extras is an extra contribution for this example being considered
147         """
148         return sigmoid(sum(t(example) for t in self.trees)+extra)
149
150     def leaf_value(self, egs, domain=[0,1]):
151         """value at the leaves for examples egs
152         domain argument is ignored"""
153         predActs = [(self.gtb_predictor(e),self.target(e)) for e in egs]
154         return sum(a-p for (p,a) in predActs) / (sum(p*(1-p) for (p,a) in
155             predActs)+self.lambda_reg)
156
157     def sum_losses(self, data_subset):
158         """returns sum of losses for dataset (assuming a leaf is formed
159         with no more splits)
160         """
161         leaf_val = self.leaf_value(data_subset)
162         error = sum(Evaluate.log_loss(self.gtb_predictor(e,leaf_val),
163             self.target(e))
164             for e in data_subset) + self.gamma
165     return error

```

Testing

```

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

```

Exercise 7.15 Find better hyperparameter settings than the default ones. Compare prediction error with other methods for Boolean datasets.

Neural Networks and Deep Learning

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

We have followed the naming conventions of Keras for the parameters: any parameters that are the same as in Keras have the same names.

8.1 Layers

A neural network is built from layers. In AIPython, unlike Keras and PyTorch, activation functions are treated as separate layers, which makes them more modular and the code more readable.

This provides a modular implementation of layers. Layers can easily be stacked in many configurations. A layer needs to implement a 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

```

8.1.1 Linear Layer

A linear layer maintains an array of weights. `self.weights[o][i]` is the weight between input i and output o . 1 is added as an extra value 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 by default initializes 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)
65                         if inf < 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)]
70
71     def output_values(self, input_values, training=False):
72         """Returns the outputs for the input values.
73         It remembers the values for the backprop.
74
75         Note in self.weights there is a weight list for every output,
76         so wts in self.weights loops over the outputs.
77         The bias is the *last* value of each list in self.weights.
78         """
79         self.inputs = input_values + [1]
80         return [sum(w*val for (w,val) in zip(wts,self.inputs))
81                 for wts in self.weights]
82
83     def backprop(self, errors):
84         """Backpropagate the errors, updating the weights and returning the
85         error in its inputs.
86         """
87         input_errors = [0]*(self.num_inputs+1)
88         for out in range(self.num_outputs):
89             for inp in range(self.num_inputs+1):
90                 input_errors[inp] += self.weights[out][inp] * errors[out]
91                 self.delta[out][inp] += self.inputs[inp] * errors[out]
92         return input_errors[:-1] # remove the error for the "1"
93
94     def update(self):
95         """updates parameters after a batch"""
96         batch_step_size = self.nn.learning_rate / self.nn.batch_size
97         for out in range(self.num_outputs):
98             for inp in range(self.num_inputs+1):

```

```

98         self.weights[out][inp] -= batch_step_size *
          self.delta[out][inp]
99         self.delta[out][inp] = 0

```

8.1.2 ReLU Layer

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

```

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

```

8.1.3 Sigmoid Layer

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

```

learnNN.py — (continued)
120 class Sigmoid_layer(Layer):
121     """sigmoids of the inputs.
122     The number of outputs is equal to the number of inputs.
123     Each output is the sigmoid of its corresponding input.
124     """
125     def __init__(self, nn):
126         Layer.__init__(self, nn)
127
128     def output_values(self, input_values, training=False):
129         """Returns the outputs for the input values.
130         It remembers the output values for the backprop.
131         """
132         self.outputs= [sigmoid(inp) for inp in input_values]
133         return self.outputs

```



```

134
135     def backprop(self, errors):
136         """Returns the derivative of the errors"""
137         return [e*out*(1-out) for e,out in zip(errors, self.outputs)]

```

8.2 Feedforward Networks

```

learnNN.py — (continued)
139 class NN(Learner):
140     def __init__(self, dataset, validation_proportion = 0.1,
141                 learning_rate=0.001):
142         """Creates a neural network for a dataset,
143         layers is the list of layers
144         """
145         self.dataset = dataset
146         self.output_type = dataset.target.ftype
147         self.learning_rate = learning_rate
148         self.input_features = dataset.input_features
149         self.num_outputs = len(self.input_features)
150         validation_num = int(len(self.dataset.train)*validation_proportion)
151         if validation_num > 0:
152             random.shuffle(self.dataset.train)
153             self.validation_set = self.dataset.train[-validation_num:]
154             self.training_set = self.dataset.train[:-validation_num]
155         else:
156             self.validation_set = []
157             self.training_set = self.dataset.train
158         self.layers = []
159         self.bn = 0 # number of batches run
160
161     def add_layer(self, layer):
162         """add a layer to the network.
163         Each layer gets number of inputs from the previous layers outputs.
164         """
165         self.layers.append(layer)
166         self.num_outputs = layer.num_outputs
167
168     def predictor(self, ex):
169         """Predicts the value of the first output for example ex.
170         """
171         values = [f(ex) for f in self.input_features]
172         for layer in self.layers:
173             values = layer.output_values(values)
174         return sigmoid(values[0]) if self.output_type == "boolean" \
175             else softmax(values, self.dataset.target.frange) if
176             self.output_type == "categorical" \
177             else values[0]

```

```

177     def predictor_string(self):
178         return "not implemented"

```

The *learn* method learns the parameters of a network.

```

learnNN.py — (continued)
180     def learn(self, epochs=5, batch_size=32, num_iter = None,
181               report_each=10):
182         """Learns parameters for a neural network using stochastic gradient
183            decent.
184            epochs is the number of times through the data (on average)
185            batch_size is the maximum size of each batch
186            num_iter is the number of iterations over the batches
187                - overrides epochs if provided (allows for fractions of epochs)
188            report_each means give the errors after each multiple of that
189            iterations
190            """
191         self.batch_size = min(batch_size, len(self.training_set)) # don't
192             have batches bigger than training size
193         if num_iter is None:
194             num_iter = (epochs * len(self.training_set)) // self.batch_size
195         #self.display(0,"Batch\t", "\t".join(criterion.__doc__ for criterion
196             in Evaluate.all_criteria))
197         for i in range(num_iter):
198             batch = random.sample(self.training_set, self.batch_size)
199             for e in batch:
200                 # compute all outputs
201                 values = [f(e) for f in self.input_features]
202                 for layer in self.layers:
203                     values = layer.output_values(values, training=True)
204                 # backpropagate
205                 predicted = [sigmoid(v) for v in values] if self.output_type
206                     == "boolean" \
207                     else softmax(values) if self.output_type ==
208                         "categorical" \
209                     else values
210                 actuals = indicator(self.dataset.target(e),
211                                   self.dataset.target.frange) \
212                     if self.output_type == "categorical" \
213                     else [self.dataset.target(e)]
214                 errors = [pred-obsd for (obsd,pred) in
215                           zip(actuals,predicted)]
216                 for layer in reversed(self.layers):
217                     errors = layer.backprop(errors)
218             # Update all parameters in batch
219             for layer in self.layers:
220                 layer.update()
221             self.bn+=1
222             if (i+1)%report_each==0:
223                 self.display(0,self.bn, "\t",
224                             "\t\t".join("{:.4f}".format(

```

```

216         self.dataset.evaluate_dataset(self.validation_set,
217                                     self.predictor, criterion))
217     for criterion in Evaluate.all_criteria,
        sep="")

```

8.3 Improved Optimization

8.3.1 Momentum

```

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

```

8.3.2 RMS-Prop

```

learnNN.py — (continued)
244 class Linear_complete_layer_RMS_Prop(Linear_complete_layer):
245     """a completely connected layer"""
246     def __init__(self, nn, num_outputs, limit=None, rho=0.9, epsilon =
247                 1e-07):
248         """A completely connected linear layer.
249         nn is a neural network that the inputs come from
250         num_outputs is the number of outputs

```

```

250         max_init is the maximum value for random initialization of
           parameters
251         """
252         Linear_complete_layer.__init__(self, nn, num_outputs, limit=limit)
253         # self.weights[o][i] is the weight between input i and output o
254         self.ms = [[0 for inf in range(self.num_inputs+1)]
255                    for outf in range(self.num_outputs)]
256         self.rho = rho
257         self.epsilon = epsilon
258
259     def update(self):
260         """updates parameters after a batch"""
261         for out in range(self.num_outputs):
262             for inp in range(self.num_inputs+1):
263                 gradient = self.delta[out][inp] / self.nn.batch_size
264                 self.ms[out][inp] = self.rho*self.ms[out][inp]+ (1-self.rho)
265                     * gradient**2
266                 self.weights[out][inp] -=
267                     self.nn.learning_rate/(self.ms[out][inp]+self.epsilon)**0.5
268                     * gradient
269                 self.delta[out][inp] = 0

```

Exercise 8.1 Implement Adam [see Section 8.2.3 of Poole and Mackworth, 2023]. The implementation is slightly more complex than RMS-Prop. Try it first with the parameter settings of Keras, as reported by Poole and Mackworth [2023]. Does it matter if epsilon is inside or outside the square root? How sensitive is the performance to the parameter settings?

8.4 Dropout

Dropout is implemented as a layer.

```

learnNN.py — (continued)
268 from utilities import flip
269 class Dropout_layer(Layer):
270     """Dropout layer
271     """
272
273     def __init__(self, nn, rate=0):
274         """
275         rate is fraction of the input units to drop. 0 <= rate < 1
276         """
277         self.rate = rate
278         Layer.__init__(self, nn)
279
280     def output_values(self, input_values, training=False):
281         """Returns the outputs for the input values.
282         It remembers the input values for the backprop.
283         """
284         if training:

```

```

285         scaling = 1/(1-self.rate)
286         self.mask = [0 if flip(self.rate) else 1
287                     for _ in input_values]
288         return [x*y*scaling for (x,y) in zip(input_values, self.mask)]
289     else:
290         return input_values
291
292     def backprop(self, errors):
293         """Returns the derivative of the errors"""
294         return [x*y for (x,y) in zip(errors, self.mask)]

```

8.5 Examples

The following constructs some neural networks (most 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)
296 #data = Data_from_file('data/mail_reading.csv', target_index=-1)
297 #data = Data_from_file('data/mail_reading_consist.csv', target_index=-1)
298 data = Data_from_file('data/SPECT.csv', prob_test=0.3, target_index=0,
299                       seed=12345)
300 #data = Data_from_file('data/iris.data', prob_test=0.2, target_index=-1) #
301     150 examples approx 120 test + 30 test
302 #data = Data_from_file('data/if_x_then_y_else_z.csv', num_train=8,
303     target_index=-1) # not linearly sep
304 #data = Data_from_file('data/holiday.csv', target_index=-1) #,
305     num_train=19)
306 #data = Data_from_file('data/processed.cleveland.data', target_index=-1)
307 #random.seed(None)
308
309 # nn3 is has a single hidden layer of width 3
310 nn3 = NN(data, validation_proportion = 0)
311 nn3.add_layer(Linear_complete_layer(nn3,3))
312 #nn3.add_layer(Sigmoid_layer(nn3))
313 nn3.add_layer(ReLU_layer(nn3))
314 nn3.add_layer(Linear_complete_layer(nn3,1)) # when using
315     output_type="boolean"
316 #nn3.learn(epochs = 100)
317
318 # nn3do is like nn3 but with dropout on the hidden layer
319 nn3do = NN(data, validation_proportion = 0)
320 nn3do.add_layer(Linear_complete_layer(nn3do,3))
321 #nn3do.add_layer(Sigmoid_layer(nn3do)) # comment this or the next
322 nn3do.add_layer(ReLU_layer(nn3do))
323 nn3do.add_layer(Dropout_layer(nn3do, rate=0.5))
324 nn3do.add_layer(Linear_complete_layer(nn3do,1))
325 #nn3do.learn(epochs = 100)

```

```

321 |
322 | # nn3_rmsp is like nn3 but uses RMS prop
323 | nn3_rmsp = NN(data, validation_proportion = 0)
324 | nn3_rmsp.add_layer(Linear_complete_layer_RMS_Prop(nn3_rmsp,3))
325 | #nn3_rmsp.add_layer(Sigmoid_layer(nn3_rmsp)) # comment this or the next
326 | nn3_rmsp.add_layer(ReLU_layer(nn3_rmsp))
327 | nn3_rmsp.add_layer(Linear_complete_layer_RMS_Prop(nn3_rmsp,1))
328 | #nn3_rmsp.learn(epochs = 100)
329 |
330 | # nn3_m is like nn3 but uses momentum
331 | mm1_m = NN(data, validation_proportion = 0)
332 | mm1_m.add_layer(Linear_complete_layer_momentum(mm1_m,3))
333 | #mm1_m.add_layer(Sigmoid_layer(mm1_m)) # comment this or the next
334 | mm1_m.add_layer(ReLU_layer(mm1_m))
335 | mm1_m.add_layer(Linear_complete_layer_momentum(mm1_m,1))
336 | #mm1_m.learn(epochs = 100)
337 |
338 | # nn2 has a single a hidden layer of width 2
339 | nn2 = NN(data, validation_proportion = 0)
340 | nn2.add_layer(Linear_complete_layer_RMS_Prop(nn2,2))
341 | nn2.add_layer(ReLU_layer(nn2))
342 | nn2.add_layer(Linear_complete_layer_RMS_Prop(nn2,1))
343 |
344 | # nn5 is has a single hidden layer of width 5
345 | nn5 = NN(data, validation_proportion = 0)
346 | nn5.add_layer(Linear_complete_layer_RMS_Prop(nn5,5))
347 | nn5.add_layer(ReLU_layer(nn5))
348 | nn5.add_layer(Linear_complete_layer_RMS_Prop(nn5,1))
349 |
350 | # nn0 has no hidden layers, and so is just logistic regression:
351 | nn0 = NN(data, validation_proportion = 0) #learning_rate=0.05)
352 | nn0.add_layer(Linear_complete_layer(nn0,1))
353 | # Or try this for RMS-Prop:
354 | #nn0.add_layer(Linear_complete_layer_RMS_Prop(nn0,1))

```

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

```

learnNN.py — (continued)
356 | from learnLinear import plot_steps
357 | from learnProblem import Evaluate
358 |
359 | # To show plots first choose a criterion to use
360 | # crit = Evaluate.log_loss
361 | # crit = Evaluate.accuracy
362 | # plot_steps(learner = nn0, data = data, criterion=crit, num_steps=10000,
      |           log_scale=False, legend_label="nn0")
363 | # plot_steps(learner = nn2, data = data, criterion=crit, num_steps=10000,
      |           log_scale=False, legend_label="nn2")

```

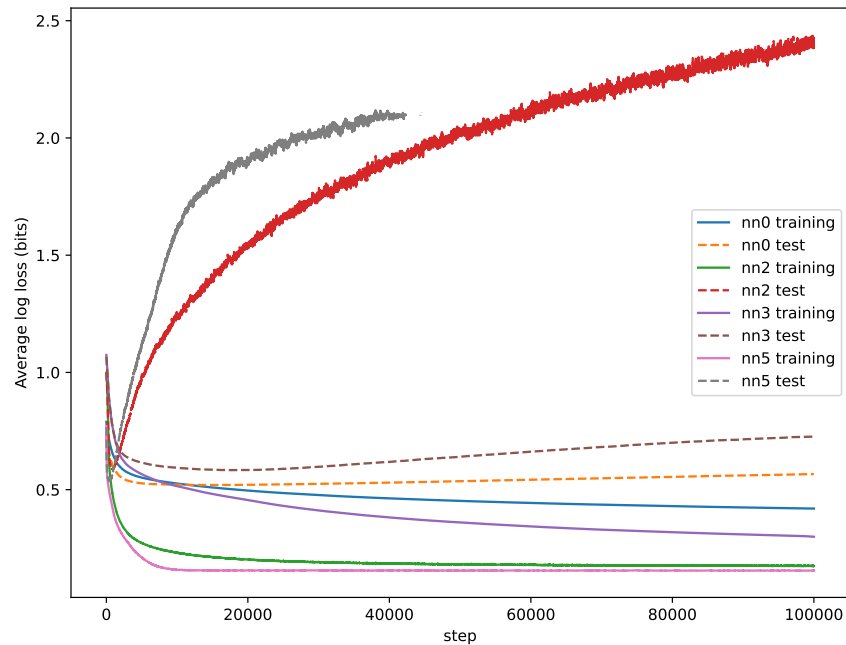


Figure 8.1: Plotting train and test log loss for various algorithms on SPECT dataset

```

364 # plot_steps(learner = nn3, data = data, criterion=crit, num_steps=10000,
      log_scale=False, legend_label="nn3")
365 # plot_steps(learner = nn5, data = data, criterion=crit, num_steps=10000,
      log_scale=False, legend_label="nn5")
366
367 # for (nn,nname) in [(nn0,"nn0"),(nn2,"nn2"),(nn3,"nn3"),(nn5,"nn5")]:
      plot_steps(learner = nn, data = data, criterion=crit,
        num_steps=100000, log_scale=False, legend_label=nname)
368
369 # Print some training examples
370 #for eg in random.sample(data.train,10): print(eg,nn3.predictor(eg))
371
372 # Print some test examples
373 #for eg in random.sample(data.test,10): print(eg,nn3.predictor(eg))
374
375 # To see the weights learned in linear layers
376 # nn3.layers[0].weights
377 # nn3.layers[2].weights
378
379 # Print test:
380 # for e in data.train: print(e,nn0.predictor(e))
381

```

```

382 def test(data, hidden_widths = [5], epochs=100,
383         optimizers = [Linear_complete_layer,
384                       Linear_complete_layer_momentum,
385                       Linear_complete_layer_RMS_Prop]):
386     data.display(0, "Batch\t", "\t".join(criterion.__doc__ for criterion in
387                                         Evaluate.all_criteria))
388     for optimizer in optimizers:
389         nn = NN(data)
390         for width in hidden_widths:
391             nn.add_layer(optimizer(nn,width))
392             nn.add_layer(ReLU_layer(nn))
393             if data.target.ftype == "boolean":
394                 nn.add_layer(optimizer(nn,1))
395             else:
396                 error(f"Not implemented: {data.output_type}")
397         nn.learn(epochs)

```

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

```

learnNN.py — (continued)
397 # Simplified version: (6000 training instances)
398 # data_mnist = Data_from_file('../MNIST/mnist_train.csv', prob_test=0.9,
399 #                             target_index=0, boolean_features=False, target_type="categorical")
400 # Full version:
401 # data_mnist = Data_from_files('../MNIST/mnist_train.csv',
402 #                               '../MNIST/mnist_test.csv', target_index=0, boolean_features=False,
403 #                               target_type="categorical")
404 # nn_mnist = NN(data_mnist, validation_proportion = 0.02,
405 #               learning_rate=0.001) #validation set = 1200
406 # nn_mnist.add_layer(Linear_complete_layer_RMS_Prop(nn_mnist,512));
407 # nn_mnist.add_layer(ReLU_layer(nn_mnist));
408 # nn_mnist.add_layer(Linear_complete_layer_RMS_Prop(nn_mnist,10))
409 # start_time = time.perf_counter();nn_mnist.learn(epochs=1,
410 #         batch_size=128);end_time = time.perf_counter();print("Time:", end_time
411 #         - start_time,"seconds") #1 epoch
412 # determine test error:
413 # data_mnist.evaluate_dataset(data_mnist.test, nn_mnist.predictor,
414 #                             Evaluate.accuracy)
415 # Print some random predictions:
416 # 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 8.2 In the definition of *nn3* above, for each of the following, first hypothesize what will happen, then test your hypothesis, then explain whether your testing confirms your hypothesis or not. Test it for more than one data set, and use more than one run for each data set.

- (a) Which fits the data better, having a sigmoid layer or a ReLU layer after the first linear layer?
- (b) Which is faster to learn, having a sigmoid layer or a ReLU layer after the first linear layer? (Hint: Plot error as a function of steps).
- (c) What happens if you have both the sigmoid layer and then a ReLU layer after the first linear layer and before the second linear layer?
- (d) What happens if you have a ReLU layer then a sigmoid layer after the first linear layer and before the second linear layer?
- (e) What happens if you have neither the sigmoid layer nor a ReLU layer after the first linear layer?

Exercise 8.3 Using some of the test set as a validation set, and stopping when the validation error gets worse, which of the settings is the best? Suggest another algorithm (either changing architecture of hyper-parameters) which you conjecture would be better. Is it better?

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

9.2 Representing Factors

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

A variable assignment, or just an **assignment**, is represented as a $\{variable : value\}$ dictionary. A factor can be evaluated when all of its variables are assigned. This is implemented in the `can_evaluate` method which can be overridden for representations that don't require all variable be assigned (such as decision trees). The method `get_value` evaluates the factor for an assignment. The assignment can include extra variables not in the factor. This method needs to be defined for every subclass.

```
_____probFactors.py — Factors for graphical models_____
11 from display import Displayable
12 import math
13
14 class Factor(Displayable):
15     nextid=0 # each factor has a unique identifier; for printing
16
```

```

17     def __init__(self, variables, name=None):
18         self.variables = variables # list of variables
19         if name:
20             self.name = name
21         else:
22             self.name = f"f{Factor.nextid}"
23             Factor.nextid += 1
24
25     def can_evaluate(self, assignment):
26         """True when the factor can be evaluated in the assignment
27         assignment is a {variable:value} dict
28         """
29         return all(v in assignment for v in self.variables)
30
31     def get_value(self, assignment):
32         """Returns the value of the factor given the assignment of values
33         to variables.
34         Needs to be defined for each subclass.
35         """
36         assert self.can_evaluate(assignment)
37         raise NotImplementedError("get_value") # abstract method

```

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

```

_____probFactors.py — (continued)_____
38     def __str__(self):
39         """returns a string representing a summary of the factor"""
40         return f"{self.name}({'.'.join(str(var) for var in
41             self.variables)})"
42
43     def to_table(self, variables=None, given={}):
44         """returns a string representation of the factor.
45         Allows for an arbitrary variable ordering.
46         variables is a list of the variables in the factor
47         (can contain other variables)"""
48         if variables==None:
49             variables = [v for v in self.variables if v not in given]
50         else: #enforce ordering and allow for extra variables in ordering
51             variables = [v for v in variables if v in self.variables and v
52                 not in given]
53         head = "\t".join(str(v) for v in variables)+"\t"+self.name
54         return head+"\n"+self.ass_to_str(variables, given, variables)
55
56     def ass_to_str(self, vars, asst, allvars):
57         #print(f"ass_to_str({vars}, {asst}, {allvars})")
58         if vars:
59             return "\n".join(self.ass_to_str(vars[1:], {**asst,
60                 vars[0]:val}, allvars)
61                 for val in vars[0].domain)

```

```

59         else:
60             val = self.get_value(asst)
61             val_st = "{:.6f}".format(val) if isinstance(val, float) else
                str(val)
62             return "\t".join(str(asst[var]) for var in allvars)
63                 + "\t"+val_st)
64
65     __repr__ = __str__

```

9.3 Conditional Probability Distributions

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

```

_____probFactors.py — (continued)_____
67 class CPD(Factor):
68     def __init__(self, child, parents):
69         """represents P(variable | parents)
70         """
71         self.parents = parents
72         self.child = child
73         Factor.__init__(self, parents+[child], name=f"Probability")
74
75     def __str__(self):
76         """A brief description of a factor using in tracing"""
77         if self.parents:
78             return f"P({self.child}|{' '.join(str(p) for p in
                self.parents)})"
79         else:
80             return f"P({self.child})"
81
82     __repr__ = __str__

```

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

```

_____probFactors.py — (continued)_____
84 class ConstantCPD(CPD):
85     def __init__(self, variable, value):
86         CPD.__init__(self, variable, [])
87         self.value = value
88     def get_value(self, assignment):
89         return 1 if self.value==assignment[self.child] else 0

```

9.3.1 Logistic Regression

A **logistic regression** CPD, for Boolean variable X represents $P(X=True \mid Y_1 \dots Y_k)$, using $k+1$ real-valued 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)_____
91 from learnLinear import sigmoid, logit
92
93 class LogisticRegression(CPD):
94     def __init__(self, child, parents, weights):
95         """A logistic regression representation of a conditional
96             probability.
97             child is the Boolean (or 0/1) variable whose CPD is being defined
98             parents is the list of parents
99             weights is list of parameters, such that weights[i+1] is the weight
100             for parents[i]
101             weights[0] is the bias.
102         """
103         assert len(weights) == 1+len(parents)
104         CPD.__init__(self, child, parents)
105         self.weights = weights
106
107     def get_value(self, assignment):
108         assert self.can_evaluate(assignment)
109         prob = sigmoid(self.weights[0]
110                       + sum(self.weights[i+1]*assignment[self.parents[i]]
111                            for i in range(len(self.parents))))
112         if assignment[self.child]: #child is true
113             return prob
114         else:
115             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 $P(Z_i \mid Y_i) = p_i$ 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)_____
115 class NoisyOR(CPD):
116     def __init__(self, child, parents, weights):

```

```

117     """A noisy representation of a conditional probability.
118     variable is the Boolean (or 0/1) child variable whose CPD is being
119         defined
120     parents is the list of Boolean (or 0/1) parents
121     weights is list of parameters, such that weights[i+1] is the weight
122         for parents[i]
123     """
124     assert len(weights) == 1+len(parents)
125     CPD.__init__(self, child, parents)
126     self.weights = weights
127
128     def get_value(self, assignment):
129         assert self.can_evaluate(assignment)
130         probfalse = (1-self.weights[0])*math.prod(1-self.weights[i+1]
131             for i in range(len(self.parents))
132             if assignment[self.parents[i]])
133         if assignment[self.child]: # child is assigned True in assignment
134             return 1-probfalse
135         else:
136             return probfalse

```

9.3.3 Tabular Factors and Prob

A **tabular factor** is a factor that represents each assignment of values to variables separately. It is represented by a Python array (or Python dict). If the variables are V_1, V_2, \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 $[0, \dots, n_i - 1]$ it can be represented as an array. Otherwise it can use a dictionary. Python is nice in that it doesn't care, whether an array or dict is used **except when enumerating the values**; enumerating a dict gives the keys (the variables) but enumerating an array gives the values. So we had to be careful not to enumerate the values.

probFactors.py — (continued)

```

136 class TabFactor(Factor):
137
138     def __init__(self, variables, values, name=None):
139         Factor.__init__(self, variables, name=name)
140         self.values = values
141
142     def get_value(self, assignment):
143         return self.get_val_rec(self.values, self.variables, assignment)
144
145     def get_val_rec(self, value, variables, assignment):
146         if variables == []:
147             return value
148         else:
149             return self.get_val_rec(value[assignment[variables[0]]],
150                                     variables[1:], assignment)

```

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

```

_____probFactors.py — (continued) _____
152 class Prob(CPD, TabFactor):
153     """A factor defined by a conditional probability table"""
154     def __init__(self, var, pars, cpt, name=None):
155         """Creates a factor from a conditional probability table, cpt
156         The cpt values are assumed to be for the ordering par+[var]
157         """
158         TabFactor.__init__(self, pars+[var], cpt, name)
159         self.child = var
160         self.parents = pars

```

9.3.4 Decision Tree Representations of Factors

A decision tree representation of a conditional probability of a child variable is either:

- `IFeq(var, val, true_cond, false_cond)` where `true_cond` and `false_cond` are decision trees. `true_cond` is used if variable `var` has value `val` in an assignment; `false_cond` is used if `var` has a different value
- a deterministic functions that has probability 1 if a parent has the same value as the child (using `SameAs(parent)`)
- a distribution over the child variable (using `Dist(dict)`).

Note that not all parents need to be assigned to evaluate the decision tree; it only needs a branch down the tree that gives the distribution.

```

_____probFactors.py — (continued) _____
162 class ProbDT(CPD):
163     def __init__(self, child, parents, dt):
164         CPD.__init__(self, child, parents)
165         self.dt = dt
166
167     def get_value(self, assignment):
168         return self.dt.get_value(assignment, self.child)
169
170     def can_evaluate(self, assignment):
171         return self.child in assignment and self.dt.can_evaluate(assignment)

```

Decision trees are made up of conditons; here equality of a value and a variable:

```

_____probFactors.py — (continued) _____
173 class IFeq:
174     def __init__(self, var, val, true_cond, false_cond):
175         self.var = var

```



```

176         self.val = val
177         self.true_cond = true_cond
178         self.false_cond = false_cond
179
180     def get_value(self, assignment, child):
181         """ IFeq(var, val, true_cond, false_cond)
182         value of true_cond is used if var has value val in assignment,
183         value of false_cond is used if var has a different value
184         """
185         if assignment[self.var] == self.val:
186             return self.true_cond.get_value(assignment, child)
187         else:
188             return self.false_cond.get_value(assignment, child)
189
190     def can_evaluate(self, assignment):
191         if self.var not in assignment:
192             return False
193         elif assignment[self.var] == self.val:
194             return self.true_cond.can_evaluate(assignment)
195         else:
196             return self.false_cond.can_evaluate(assignment)

```

The following is a deterministic function that is true if the parent has the same value as the child. This is used for deterministic conditional probabilities (as is common for causal models, as described in Chapter 11).

probFactors.py — (continued)

```

198 class SameAs:
199     def __init__(self, parent):
200         """1 when child has same value as parent, otherwise 0"""
201         self.parent = parent
202
203     def get_value(self, assignment, child):
204         return 1 if assignment[child]==assignment[self.parent] else 0
205
206     def can_evaluate(self, assignment):
207         return self.parent in assignment

```

At the leaves are distributions over the child variable.

probFactors.py — (continued)

```

209 class Dist:
210     def __init__(self, dist):
211         """Dist is an array or dictionary indexed by value of current
212         child"""
213         self.dist = dist
214
215     def get_value(self, assignment, child):
216         return self.dist[assignment[child]]
217
218     def can_evaluate(self, assignment):
219         return True

```

The following shows a decision representation of the Example 9.18 of Poole and Mackworth [2023]. When the Action is to go out, the probability is a function of rain; otherwise it is a function of full.

```

#####_probFactors.py — (continued)_____
220 ##### A decision tree representation Example 9.18 of AIFCA 3e
221 from variable import Variable
222
223 boolean = [False, True]
224
225 action = Variable('Action', ['go_out', 'get_coffee'], position=(0.5,0.8))
226 rain = Variable('Rain', boolean, position=(0.2,0.8))
227 full = Variable('Cup Full', boolean, position=(0.8,0.8))
228
229 wet = Variable('Wet', boolean, position=(0.5,0.2))
230 p_wet = ProbDT(wet,[action,rain,full],
231               IFeq(action, 'go_out',
232                     IFeq(rain, True, Dist([0.2,0.8]), Dist([0.9,0.1])),
233                     IFeq(full, True, Dist([0.4,0.6]), Dist([0.7,0.3]))))
234
235 # See probRC for wetBN which expands this example to a complete network

```

9.4 Graphical Models

A graphical model consists of a title, a set of variables, and a set of factors.

```

#####_probGraphicalModels.py — Graphical Models and Belief Networks_____
11 from display import Displayable
12 from variable import Variable
13 from probFactors import CPD, Prob
14 import matplotlib.pyplot as plt
15
16 class GraphicalModel(Displayable):
17     """The class of graphical models.
18     A graphical model consists of a title, a set of variables and a set of
19     factors.
20
21     vars is a set of variables
22     factors is a set of factors
23     """
24     def __init__(self, title, variables=None, factors=None):
25         self.title = title
26         self.variables = variables
27         self.factors = factors

```

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

tion (that all factors are conditional probabilities), and builds some useful data structures.

```

28 class BeliefNetwork(GraphicalModel):
29     """The class of belief networks."""
30
31     def __init__(self, title, variables, factors):
32         """vars is a set of variables
33         factors is a set of factors. All of the factors are instances of
34             CPD (e.g., Prob).
35             """
36         GraphicalModel.__init__(self, title, variables, factors)
37         assert all(isinstance(f,CPD) for f in factors), factors
38         self.var2cpt = {f.child:f for f in factors}
39         self.var2parents = {f.child:f.parents for f in factors}
40         self.children = {n:[] for n in self.variables}
41         for v in self.var2parents:
42             for par in self.var2parents[v]:
43                 self.children[par].append(v)
44         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.

```

45 def topological_sort(self):
46     """creates a topological ordering of variables such that the
47         parents of
48         a node are before the node.
49         """
50     if self.topological_sort_saved:
51         return self.topological_sort_saved
52     next_vars = {n for n in self.var2parents if not self.var2parents[n]}
53     self.display(3,'topological_sort: next_vars',next_vars)
54     top_order=[]
55     while next_vars:
56         var = next_vars.pop()
57         self.display(3,'select variable',var)
58         top_order.append(var)
59         next_vars -= {ch for ch in self.children[var]
60                     if all(p in top_order for p in
61                         self.var2parents[ch])}
62         self.display(3,'var_with_no_parents_left',next_vars)
63     self.display(3,"top_order",top_order)
64     assert
65         set(top_order)==set(self.var2parents), (top_order,self.var2parents)
66     self.topologicalsort_saved=top_order
67     return top_order

```

4-chain

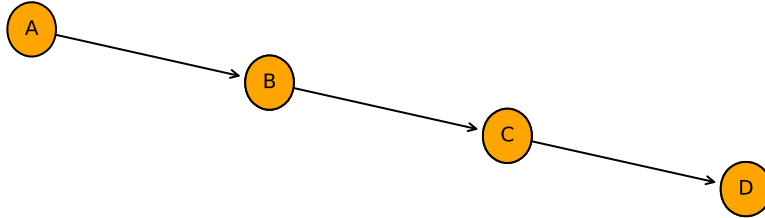


Figure 9.1: bn_4ch.show()

9.4.1 Showing Belief Networks

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

```

66     def show(self, fontsize=10, facecolor='orange'):
67         plt.ion() # interactive
68         ax = plt.figure().gca()
69         ax.set_axis_off()
70         plt.title(self.title, fontsize=fontsize)
71         bbox =
72             dict(boxstyle="round4,pad=1.0,rounding_size=0.5",facecolor=facecolor)
73         for var in self.variables: #reversed(self.topological_sort()):
74             for par in self.var2parents[var]:
75                 ax.annotate(var.name, par.position, xytext=var.position,
76                             arrowprops={'arrowstyle':'<-'},bbox=bbox,
77                             ha='center', va='center',
78                             fontsize=fontsize)
79         for var in self.variables:
80             x,y = var.position
81             plt.text(x,y,var.name,bbox=bbox,ha='center', va='center',
82                     fontsize=fontsize)

```

9.4.2 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.

```

81 ##### Simple Example Used for Unit Tests #####

```

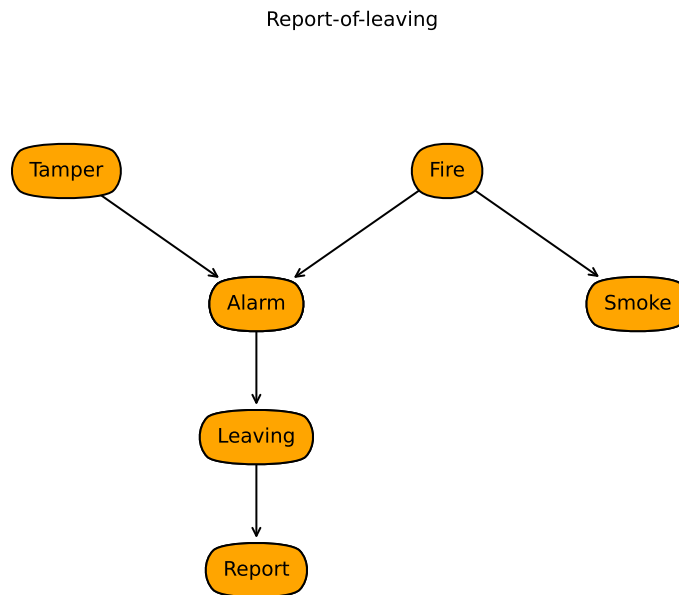


Figure 9.2: The report-of-leaving belief network

```

82 | boolean = [False, True]
83 | A = Variable("A", boolean, position=(0,0.8))
84 | B = Variable("B", boolean, position=(0.333,0.7))
85 | C = Variable("C", boolean, position=(0.666,0.6))
86 | D = Variable("D", boolean, position=(1,0.5))
87 |
88 | f_a = Prob(A,[],[0.4,0.6])
89 | f_b = Prob(B,[A],[[0.9,0.1],[0.2,0.8]])
90 | f_c = Prob(C,[B],[[0.6,0.4],[0.3,0.7]])
91 | f_d = Prob(D,[C],[[0.1,0.9],[0.75,0.25]])
92 |
93 | bn_4ch = BeliefNetwork("4-chain", {A,B,C,D}, {f_a,f_b,f_c,f_d})

```

Report-of-Leaving Example

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

```

_____probExamples.py — Example belief networks_____
11 | from variable import Variable
12 | from probFactors import CPD, Prob, LogisticRegression, NoisyOR, ConstantCPD
13 | from probGraphicalModels import BeliefNetwork
14 |

```

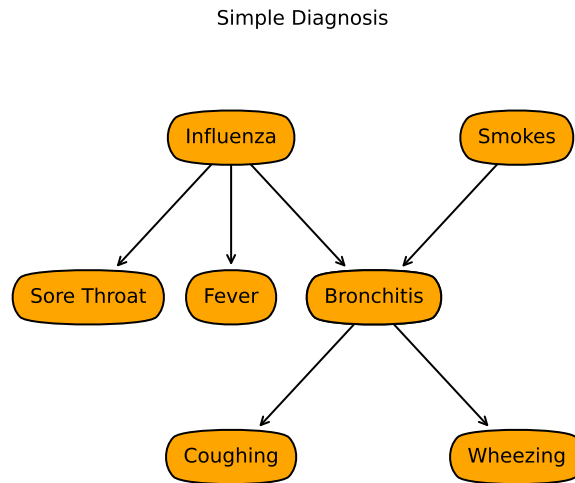


Figure 9.3: Simple diagnosis example; `simple_diagnosis.show()`

```

15 # Belief network report-of-leaving example (Example 9.13 shown in Figure
    9.3) of
16 # Poole and Mackworth, Artificial Intelligence, 2023 http://artint.info
17 boolean = [False, True]
18
19 Alarm = Variable("Alarm", boolean, position=(0.366,0.5))
20 Fire = Variable("Fire", boolean, position=(0.633,0.75))
21 Leaving = Variable("Leaving", boolean, position=(0.366,0.25))
22 Report = Variable("Report", boolean, position=(0.366,0.0))
23 Smoke = Variable("Smoke", boolean, position=(0.9,0.5))
24 Tamper = Variable("Tamper", boolean, position=(0.1,0.75))
25
26 f_ta = Prob(Tamper,[],[0.98,0.02])
27 f-fi = Prob(Fire,[],[0.99,0.01])
28 f-sm = Prob(Smoke,[Fire],[[0.99,0.01],[0.1,0.9]])
29 f-al = Prob(Alarm,[Fire,Tamper],[[0.9999, 0.0001], [0.15, 0.85]], [[0.01,
    0.99], [0.5, 0.5]])
30 f-lv = Prob(Leaving,[Alarm],[[0.999, 0.001], [0.12, 0.88]])
31 f-re = Prob(Report,[Leaving],[[0.99, 0.01], [0.25, 0.75]])
32
33 bn_report = BeliefNetwork("Report-of-leaving",
    {Tamper,Fire,Smoke,Alarm,Leaving,Report},
34 {f_ta,f-fi,f-sm,f-al,f-lv,f-re})
  
```

Simple Diagnostic Example

This is the “simple diagnostic example” of Exercise 9.1 of Poole and Mackworth [2023], reproduced here as Figure 9.3

probExamples.py — (continued) —

```

36 # Belief network simple-diagnostic example (Exercise 9.3 shown in Figure
    9.39) of
37 # Poole and Mackworth, Artificial Intelligence, 2023 http://artint.info
38
39 Influenza = Variable("Influenza", boolean, position=(0.4,0.8))
40 Smokes = Variable("Smokes", boolean, position=(0.8,0.8))
41 SoreThroat = Variable("Sore Throat", boolean, position=(0.2,0.5))
42 HasFever = Variable("Fever", boolean, position=(0.4,0.5))
43 Bronchitis = Variable("Bronchitis", boolean, position=(0.6,0.5))
44 Coughing = Variable("Coughing", boolean, position=(0.4,0.2))
45 Wheezing = Variable("Wheezing", boolean, position=(0.8,0.2))
46
47 p_infl = Prob(Influenza,[],[0.95,0.05])
48 p_smokes = Prob(Smokes,[],[0.8,0.2])
49 p_sth = Prob(SoreThroat,[Influenza],[[0.999,0.001],[0.7,0.3]])
50 p_fever = Prob(HasFever,[Influenza],[[0.99,0.05],[0.9,0.1]])
51 p_bronc = Prob(Bronchitis,[Influenza,Smokes],[[0.9999, 0.0001], [0.3,
    0.7]], [[0.1, 0.9], [0.01, 0.99]])
52 p_cough = Prob(Coughing,[Bronchitis],[[0.93,0.07],[0.2,0.8]])
53 p_wheeze = Prob(Wheezing,[Bronchitis],[[0.999,0.001],[0.4,0.6]])
54
55 simple_diagnosis = BeliefNetwork("Simple Diagnosis",
56                                 {Influenza, Smokes, SoreThroat, HasFever, Bronchitis,
57                                  Coughing, Wheezing},
58                                 {p_infl, p_smokes, p_sth, p_fever, p_bronc, p_cough,
59                                  p_wheeze})

```

Sprinkler Example

The third belief network is the sprinkler example from Pearl [2009]. The output of `bn_sprinkler.show()` is shown in Figure 9.4 of this document.

```

probExamples.py — (continued)
59 Season = Variable("Season", ["dry_season","wet_season"],
    position=(0.5,0.9))
60 Sprinkler = Variable("Sprinkler", ["on","off"], position=(0.9,0.6))
61 Rained = Variable("Rained", boolean, position=(0.1,0.6))
62 Grass_wet = Variable("Grass wet", boolean, position=(0.5,0.3))
63 Grass_shiny = Variable("Grass shiny", boolean, position=(0.1,0))
64 Shoes_wet = Variable("Shoes wet", boolean, position=(0.9,0))
65
66 f_season = Prob(Season,[],{'dry_season':0.5, 'wet_season':0.5})
67 f_sprinkler = Prob(Sprinkler,[Season],{'dry_season':{'on':0.4,'off':0.6},
68                                         'wet_season':{'on':0.01,'off':0.99}})
69 f_rained = Prob(Rained,[Season],{'dry_season':[0.9,0.1], 'wet_season':
    [0.2,0.8]})
70 f_wet = Prob(Grass_wet,[Sprinkler,Rained], {'on': [[0.1,0.9],[0.01,0.99]],
71                                         'off':[[0.99,0.01],[0.3,0.7]]})
72 f_shiny = Prob(Grass_shiny, [Grass_wet], [[0.95,0.05], [0.3,0.7]])
73 f_shoes = Prob(Shoes_wet, [Grass_wet], [[0.98,0.02], [0.35,0.65]])

```

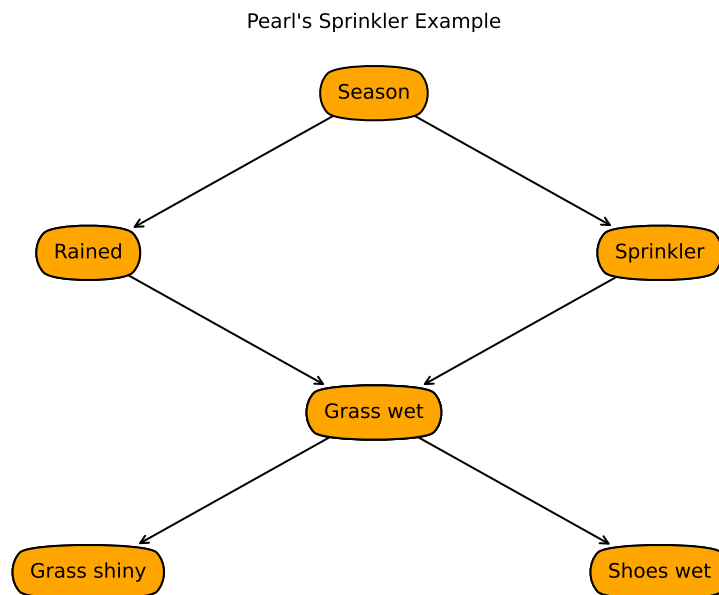


Figure 9.4: The sprinkler belief network

```

74 |
75 | bn_sprinkler = BeliefNetwork("Pearl's Sprinkler Example",
76 |                             {Season, Sprinkler, Rained, Grass_wet, Grass_shiny,
77 |                               Shoes_wet},
78 |                             {f_season, f_sprinkler, f_rained, f_wet, f_shiny,
79 |                               f_shoes})

```

Bipartite Diagnostic Model with Noisy-or

The belief network `bn_no1` is a bipartite diagnostic model, with independent diseases, and the symptoms depend on the diseases, where the CPDs are defined using noisy-or. Bipartite means it is in two parts; the diseases are only connected to the symptoms and the symptoms are only connected to the diseases. The output of `bn_no1.show()` is shown in Figure 9.5 of this document.

```

##### probExamples.py — (continued) #####
79 | ##### Bipartite Diagnostic Network #####
80 | Cough = Variable("Cough", boolean, (0.1,0.1))
81 | Fever = Variable("Fever", boolean, (0.5,0.1))
82 | Sneeze = Variable("Sneeze", boolean, (0.9,0.1))
83 | Cold = Variable("Cold",boolean, (0.1,0.9))
84 | Flu = Variable("Flu",boolean, (0.5,0.9))

```

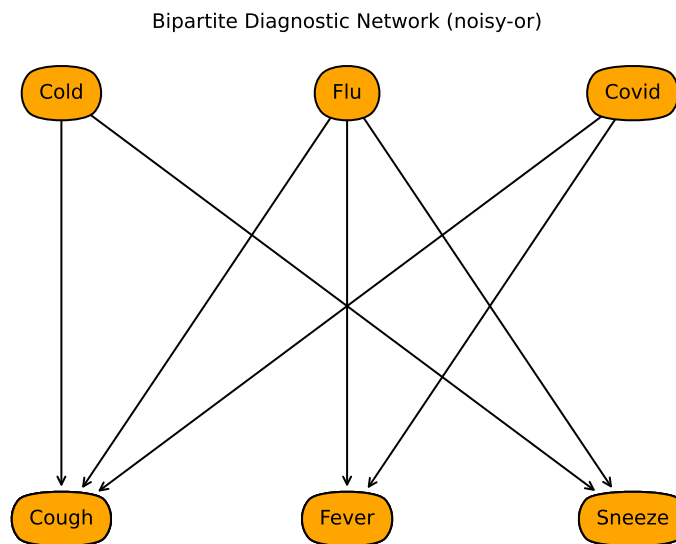



Figure 9.5: A bipartite diagnostic network

```

85 Covid = Variable("Covid",boolean, (0.9,0.9))
86
87 p_cold_no = Prob(Cold,[],[0.9,0.1])
88 p_flu_no = Prob(Flu,[],[0.95,0.05])
89 p_covid_no = Prob(Covid,[],[0.99,0.01])
90
91 p_cough_no = NoisyOR(Cough, [Cold,Flu,Covid], [0.1, 0.3, 0.2, 0.7])
92 p_fever_no = NoisyOR(Fever, [Flu,Covid], [0.01, 0.6, 0.7])
93 p_sneeze_no = NoisyOR(Sneeze, [Cold,Flu ], [0.05, 0.5, 0.2 ])
94
95 bn_no1 = BeliefNetwork("Bipartite Diagnostic Network (noisy-or)",
96                        {Cough, Fever, Sneeze, Cold, Flu, Covid},
97                        {p_cold_no, p_flu_no, p_covid_no, p_cough_no,
98                          p_fever_no, p_sneeze_no})
99
100 # to see the conditional probability of Noisy-or do:
101 # print(p_cough_no.to_table())
102
103 # example from box "Noisy-or compared to logistic regression"
104 # X = Variable("X",boolean)
105 # w0 = 0.01
106 # print(NoisyOR(X,[A,B,C,D],[w0, 1-(1-0.05)/(1-w0), 1-(1-0.1)/(1-w0),
107                    1-(1-0.2)/(1-w0), 1-(1-0.2)/(1-w0), ]).to_table(given={X:True}))

```

Bipartite Diagnostic Model with Logistic Regression

The belief network `bn_lr1` is a bipartite diagnostic model, with independent diseases, and the symptoms depend on the diseases, where the CPDs are defined using logistic regression. It has the same graphical structure as the previous example (see Figure 9.5). This has the (approximately) the same conditional probabilities as the previous example when zero or one diseases are present. Note that $\text{sigmoid}(-2.2) \approx 0.1$

```

107
108 p_cold_lr = Prob(Cold,[],[0.9,0.1])
109 p_flu_lr = Prob(Flu,[],[0.95,0.05])
110 p_covid_lr = Prob(Covid,[],[0.99,0.01])
111
112 p_cough_lr = LogisticRegression(Cough, [Cold,Flu,Covid], [-2.2, 1.67,
113                                     1.26, 3.19])
114 p_fever_lr = LogisticRegression(Fever, [Flu,Covid], [-4.6, 5.02,
115                                     5.46])
116 p_sneeze_lr = LogisticRegression(Sneeze, [Cold,Flu ], [-2.94, 3.04, 1.79
117                                     ])
118
119 bn_lr1 = BeliefNetwork("Bipartite Diagnostic Network - logistic
120                        regression",
121                        {Cough, Fever, Sneeze, Cold, Flu, Covid},
122                        {p_cold_lr, p_flu_lr, p_covid_lr, p_cough_lr,
123                          p_fever_lr, p_sneeze_lr})
124
125 # to see the conditional probability of Noisy-or do:
126 #print(p_cough_lr.to_table())
127
128 # example from box "Noisy-or compared to logistic regression"
129 # from learnLinear import sigmoid, logit
130 # w0=logit(0.01)
131 # X = Variable("X",boolean)
132 # print(LogisticRegression(X,[A,B,C,D],[w0, logit(0.05)-w0, logit(0.1)-w0,
133 #                               logit(0.2)-w0, logit(0.2)-w0]).to_table(given={X:True}))
134 # try to predict what would happen (and then test) if we had
135 # w0=logit(0.01)

```

9.5 Inference Methods

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

```

probGraphicalModels.py — (continued)

```

```

95 from display import Displayable
96
97 class InferenceMethod(Displayable):
98     """The abstract class of graphical model inference methods"""
99     method_name = "unnamed" # each method should have a method name
100
101     def __init__(self, gm=None):
102         self.gm = gm
103
104     def query(self, qvar, obs={}):
105         """returns a {value:prob} dictionary for the query variable"""
106         raise NotImplementedError("InferenceMethod query") # abstract method

```

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

```

_____probGraphicalModels.py — (continued)_____
108     def testIM(self, threshold=0.000000001):
109         solver = self(bn_4ch)
110         res = solver.query(B, {D: True})
111         correct_answer = 0.429632380245
112         assert correct_answer - threshold < res[True] <
113             correct_answer + threshold, \
114             f"value {res[True]} not in desired range for
115             {self.method_name}"
116         print(f"Unit test passed for {self.method_name}.")

```

9.5.1 Showing Posterior Distributions

The `show_post` method draws the posterior distribution of all variables. Figure 9.6 shows the result of `bn_reportRC.show_post({Report: True})` when run after loading `probRC.py` (see below).

```

_____probGraphicalModels.py — (continued)_____
116     def show_post(self, obs={}, num_format="{:.3f}", fontsize=10,
117         facecolor='orange'):
118         """draws the graphical model conditioned on observations obs
119         num_format is number format (allows for more or less precision)
120         fontsize gives size of the text
121         facecolor gives the color of the nodes
122         """
123         plt.ion() # interactive
124         ax = plt.figure().gca()
125         ax.set_axis_off()
126         plt.title(self.gm.title + " observed: " + str(obs), fontsize=fontsize)
127         bbox = dict(boxstyle="round4", pad=1.0, rounding_size=0.5,
128             facecolor=facecolor)
129         vartext = {} # variable:text dictionary
130         for var in self.gm.variables: #reversed(self.gm.topological_sort()):

```

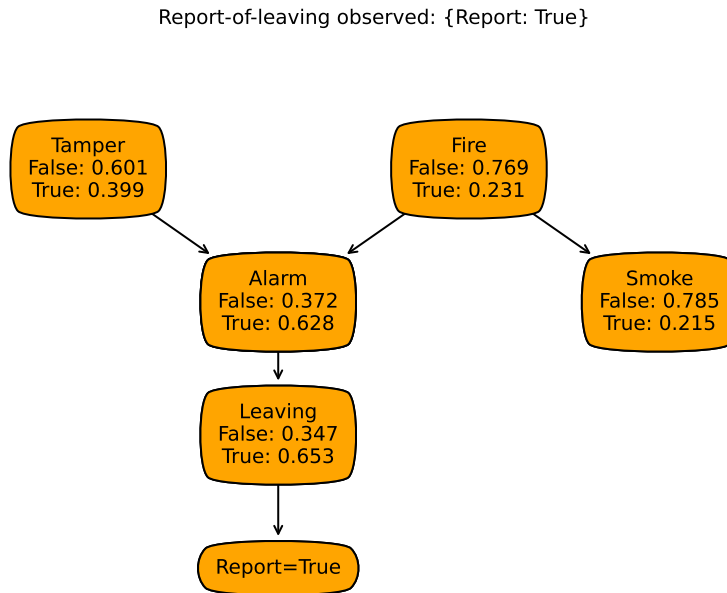


Figure 9.6: The report-of-leaving belief network with posterior distributions

```

129     if var in obs:
130         text = var.name + "=" + str(obs[var])
131     else:
132         distn = self.query(var, obs=obs)
133
134         text = var.name + "\n" + "\n".join(str(d)+"":
135             "+num_format.format(v) for (d,v) in distn.items()")
136     vartext[var] = text
137     # Draw arcs
138     for par in self.gm.var2parents[var]:
139         ax.annotate(text, par.position, xytext=var.position,
140             arrowprops={'arrowstyle': '<-'}, bbox=bbox,
141             ha='center', va='center',
142             fontsize=fontsize)
143     for var in self.gm.variables:
144         x,y = var.position
145         plt.text(x,y,vartext[var], bbox=bbox, ha='center', va='center',
146             fontsize=fontsize)

```

9.6 Naive Search

An instance of a *ProbSearch* object takes in a graphical model. The query method uses naive search to compute the probability of a query variable given obser-

uations on other variables. See Figure 9.9 of Poole and Mackworth [2023].

```

-----probRC.py — Search-based Inference 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 search
17
18     gm is graphical model to query
19     """
20     method_name = "naive search"
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)
34                     for val in qvar.domain}
35         else:
36             if split_order == None:
37                 split_order = [v for v in self.gm.variables
38                               if (v not in obs) and v != qvar]
39             unnorm = [self.prob_search({qvar:val}|obs, self.gm.factors,
40                                     split_order)
41                       for val in qvar.domain]
42             p_obs = sum(unnorm)
43             return {val:pr/p_obs for val,pr in zip(qvar.domain, unnorm)}

```

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

```

-----probRC.py — (continued)-----
44 def prob_search(self, context, factors, split_order):
45     """simple search algorithm
46     context: a variable:value dictionary
47     factors: a set of factors
48     split_order: list of variables not assigned in context
49     returns sum over variable assignments to variables in split order
50     of product of factors """
51     self.display(2, "calling prob_search,", (context, factors, split_order))

```

```

51     if not factors:
52         return 1
53     elif to_eval := {fac for fac in factors
54                     if fac.can_evaluate(context)}:
55         # evaluate factors when all variables are assigned
56         self.display(3, "prob_search evaluating factors", to_eval)
57         val = math.prod(fac.get_value(context) for fac in to_eval)
58         return val * self.prob_search(context, factors-to_eval,
59                                     split_order)
59     else:
60         total = 0
61         var = split_order[0]
62         self.display(3, "prob_search branching on", var)
63         for val in var.domain:
64             total += self.prob_search({var:val}|context, factors,
65                                     split_order[1:])
66         self.display(3, "prob_search branching on", var, "returning",
67                     total)
68         return total

```

9.7 Recursive Conditioning

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

The cache is initialized with the empty context and empty factors has probability 1. This means that checking the cache can act as the base case when the context is empty.

```

probRC.py — (continued)
68 class ProbRC(ProbSearch):
69     method_name = "recursive conditioning"
70
71     def __init__(self, gm=None):
72         self.cache = {(frozenset(), frozenset()): 1}
73         ProbSearch.__init__(self, gm)
74
75     def prob_search(self, context, factors, split_order):
76         """ returns sum_{split_order} prod_{factors} given assignment in
77             context
78             context is a variable:value dictionary
79             factors is a set of factors
80             split_order: list of variables in factors that are not in context
81             """
82         self.display(3, "calling rc,", (context, factors))
83         ce = (frozenset(context.items()), frozenset(factors)) # key for the
84                 cache entry

```

```

83     if ce in self.cache:
84         self.display(3, "rc cache lookup", (context, factors))
85         return self.cache[ce]
86     elif vars_not_in_factors := {var for var in context
87                                 if not any(var in fac.variables
88                                             for fac in factors)}:
89         # forget variables not in any factor
90         self.display(3, "rc forgetting variables", vars_not_in_factors)
91         return self.prob_search({key:val for (key,val) in
92                                 context.items()
93                                 if key not in vars_not_in_factors},
94                                 factors, split_order)
95     elif to_eval := {fac for fac in factors
96                     if fac.can_evaluate(context)}:
97         # evaluate factors when all variables are assigned
98         self.display(3, "rc evaluating factors", to_eval)
99         val = math.prod(fac.get_value(context) for fac in to_eval)
100        if val == 0:
101            return 0
102        else:
103            return val * self.prob_search(context,
104                                          {fac for fac in factors
105                                          if fac not in to_eval},
106                                          split_order)
107    elif len(comp := connected_components(context, factors,
108                                          split_order)) > 1:
109        # there are disconnected components
110        self.display(3, "splitting into connected components", comp, "in
111        context", context)
112        return (math.prod(self.prob_search(context, f, eo) for (f, eo) in
113                          comp))
114    else:
115        assert split_order, "split_order should not be empty to get
116        here"
117        total = 0
118        var = split_order[0]
119        self.display(3, "rc branching on", var)
120        for val in var.domain:
121            total += self.prob_search({var:val}|context, factors,
122                                     split_order[1:])
123        self.cache[ce] = total
124        self.display(2, "rc branching on", var, "returning", total)
125        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 compo-

nent 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

```

121 def connected_components(context, factors, split_order):
122     """returns a list of (f,e) where f is a subset of factors and e is a
123     subset of split_order
124     such that each element shares the same variables that are disjoint from
125     other elements.
126     """
127     other_factors = set(factors) #copies factors
128     factors_to_check = {other_factors.pop()} # factors in connected
129     component still to be checked
130     component_factors = set() # factors in first connected component
131     already checked
132     component_variables = set() # variables in first connected component
133     while factors_to_check:
134         next_fac = factors_to_check.pop()
135         component_factors.add(next_fac)
136         new_vars = set(next_fac.variables) - component_variables -
137         context.keys()
138         component_variables |= new_vars
139         for var in new_vars:
140             factors_to_check |= {f for f in other_factors
141                                if var in f.variables}
142             other_factors -= factors_to_check # set difference
143     if other_factors:
144         return ( [(component_factors,[e for e in split_order
145                                if e in component_variables])]
146                 + connected_components(context, other_factors,
147                                     [e for e in split_order
148                                     if e not in component_variables]) )
149     else:
150         return [(component_factors, split_order)]

```

Testing:

```

147 from probGraphicalModels import bn_4ch, A,B,C,D,f_a,f_b,f_c,f_d
148 bn_4chv = ProbRC(bn_4ch)
149 ## bn_4chv.query(A,{})
150 ## bn_4chv.query(D,{})
151 ## InferenceMethod.max_display_level = 3 # show more detail in displaying
152 ## InferenceMethod.max_display_level = 1 # show less detail in displaying

```



```

153 ## bn_4chv.query(A,{D:True},[C,B])
154 ## bn_4chv.query(B,{A:True,D:False})
155
156 from probExamples import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
157 bn_reportRC = ProbRC(bn_report) # answers queries using recursive
    conditioning
158 ## bn_reportRC.query(Tamper,{})
159 ## InferenceMethod.max_display_level = 0 # show no detail in displaying
160 ## bn_reportRC.query(Leaving,{})
161 ## bn_reportRC.query(Tamper,{},
    split_order=[Smoke,Fire,Alarm,Leaving,Report])
162 ## bn_reportRC.query(Tamper,{Report:True})
163 ## bn_reportRC.query(Tamper,{Report:True,Smoke:False})
164
165 ## To display resulting posteriors try:
166 # bn_reportRC.show_post({})
167 # bn_reportRC.show_post({Smoke:False})
168 # bn_reportRC.show_post({Report:True})
169 # bn_reportRC.show_post({Report:True, Smoke:False})
170
171 ## Note what happens to the cache when these are called in turn:
172 ## bn_reportRC.query(Tamper,{Report:True},
    split_order=[Smoke,Fire,Alarm,Leaving])
173 ## bn_reportRC.query(Smoke,{Report:True},
    split_order=[Tamper,Fire,Alarm,Leaving])
174
175 from probExamples import bn_sprinkler, Season, Sprinkler, Rained,
    Grass_wet, Grass_shiny, Shoes_wet
176 bn_sprinklerv = ProbRC(bn_sprinkler)
177 ## bn_sprinklerv.query(Shoes_wet,{})
178 ## bn_sprinklerv.query(Shoes_wet,{Rained:True})
179 ## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:True})
180 ## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:False,Rained:True})
181
182 from probExamples import bn_no1, bn_lr1, Cough, Fever, Sneeze, Cold, Flu,
    Covid
183 bn_no1v = ProbRC(bn_no1)
184 bn_lr1v = ProbRC(bn_lr1)
185 ## bn_no1v.query(Flu, {Fever:1, Sneeze:1})
186 ## bn_lr1v.query(Flu, {Fever:1, Sneeze:1})
187 ## bn_lr1v.query(Cough,{})
188 ## bn_lr1v.query(Cold,{Cough:1,Sneeze:0,Fever:1})
189 ## bn_lr1v.query(Flu,{Cough:0,Sneeze:1,Fever:1})
190 ## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1})
191 ## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:0})
192 ## bn_lr1v.query(Covid,{Cough:1,Sneeze:0,Fever:1,Flu:1})
193
194 if __name__ == "__main__":
195     InferenceMethod.testIM(ProbSearch)
196     InferenceMethod.testIM(ProbRC)

```

The following example uses the decision tree representation of Section 9.3.4 (page 210).

```

probRC.py — (continued)
198 from probFactors import Prob, action, rain, full, wet, p_wet
199 from probGraphicalModels import BeliefNetwork
200 p_action = Prob(action,[],{'go_out':0.3, 'get_coffee':0.7})
201 p_rain = Prob(rain,[],[0.4,0.6])
202 p_full = Prob(full,[],[0.1,0.9])
203
204 wetBN = BeliefNetwork("Wet (decision tree CPD)", {action, rain, full, wet},
205                      {p_action, p_rain, p_full, p_wet})
206 wetRC = ProbRC(wetBN)
207 # wetRC.query(wet, {action:'go_out', rain:True})
208 # wetRC.show_post({action:'go_out', rain:True})
209 # wetRC.show_post({action:'go_out', wet:True})

```

Exercise 9.1 Does recursive conditioning split on variable `full` for the query commented out above? Does it need to? Fix the code so that decision tree representations of conditional probabilities can be evaluated as soon as possible.

9.8 Variable Elimination

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

```

probVE.py — Variable Elimination for Graphical Models
11 from probFactors import Factor, FactorObserved, FactorSum, factor_times
12 from probGraphicalModels import GraphicalModel, InferenceMethod
13
14 class VE(InferenceMethod):
15     """The class that queries Graphical Models using variable elimination.
16
17     gm is graphical model to query
18     """
19     method_name = "variable elimination"
20
21     def __init__(self, gm=None):
22         InferenceMethod.__init__(self, gm)
23
24     def query(self, var, obs={}, elim_order=None):
25         """computes P(var|obs) where
26         var is a variable
27         obs is a {variable:value} dictionary"""
28         if var in obs:
29             return {var:1 if val == obs[var] else 0 for val in var.domain}
30         else:
31             if elim_order == None:

```

```

32         elim_order = self.gm.variables
33         projFactors = [self.project_observations(fact,obs)
34                        for fact in self.gm.factors]
35         for v in elim_order:
36             if v != var and v not in obs:
37                 projFactors = self.eliminate_var(projFactors,v)
38         unnorm = factor_times(var,projFactors)
39         p_obs=sum(unnorm)
40         self.display(1,"Unnormalized probs:",unnorm,"Prob obs:",p_obs)
41         return {val:pr/p_obs for val,pr in zip(var.domain, unnorm)}

```

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

```

_____probFactors.py — (continued)_____
237 class FactorObserved(Factor):
238     def __init__(self,factor,obs):
239         Factor.__init__(self, [v for v in factor.variables if v not in obs])
240         self.observed = obs
241         self.orig_factor = factor
242
243     def get_value(self,assignment):
244         return self.orig_factor.get_value(assignment|self.observed)

```

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

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

We store the values in a list in a lazy manner; if they are already computed, we used the stored values. If they are not already computed we can compute and store them.

```

_____probFactors.py — (continued)_____
246 class FactorSum(Factor):
247     def __init__(self,var,factors):
248         self.var_summed_out = var
249         self.factors = factors
250         vars = list({v for fac in factors
251                     for v in fac.variables if v is not var})
252         #for fac in factors:
253         #    for v in fac.variables:
254         #        if v is not var and v not in vars:
255         #            vars.append(v)
256         Factor.__init__(self,vars)
257         self.values = {}
258
259     def get_value(self,assignment):

```

```

260         """lazy implementation: if not saved, compute it. Return saved
           value"""
261         asst = frozenset(assignment.items())
262         if asst in self.values:
263             return self.values[asst]
264         else:
265             total = 0
266             new_asst = assignment.copy()
267             for val in self.var_summed_out.domain:
268                 new_asst[self.var_summed_out] = val
269                 total += math.prod(fac.get_value(new_asst) for fac in
                                     self.factors)
270             self.values[asst] = total
271         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)
273 def factor_times(variable, factors):
274     """when factors are factors just on variable (or on no variables)"""
275     prods = []
276     facs = [f for f in factors if variable in f.variables]
277     for val in variable.domain:
278         ast = {variable:val}
279         prods.append(math.prod(f.get_value(ast) for f in facs))
280     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 of 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 probExamples import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
86 bn_reportv = VE(bn_report) # answers queries using variable elimination
87 ## bn_reportv.query(Tamper,{})
88 ## InferenceMethod.max_display_level = 0 # show no detail in displaying
89 ## bn_reportv.query(Leaving,{})
90 ## bn_reportv.query(Tamper,{},elim_order=[Smoke,Report,Leaving,Alarm,Fire])
91 ## bn_reportv.query(Tamper,{Report:True})
92 ## bn_reportv.query(Tamper,{Report:True,Smoke:False})
93
94 from probExamples import bn_sprinkler, Season, Sprinkler, Rained,
95     Grass_wet, Grass_shiny, Shoes_wet
96 bn_sprinklerv = VE(bn_sprinkler)
97 ## bn_sprinklerv.query(Shoes_wet,{})
98 ## bn_sprinklerv.query(Shoes_wet,{Rained:True})
99 ## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:True})
100 ## bn_sprinklerv.query(Shoes_wet,{Grass_shiny:False,Rained:True})
101
102 from probExamples import bn_lr1, Cough, Fever, Sneeze, Cold, Flu, Covid
103 vediag = VE(bn_lr1)
104 ## vediag.query(Cough,{})
105 ## vediag.query(Cold,{Cough:1,Sneeze:0,Fever:1})
106 ## vediag.query(Flu,{Cough:0,Sneeze:1,Fever:1})
107 ## vediag.query(Covid,{Cough:1,Sneeze:0,Fever:1})

```

```

107 ## vediag.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 (possibly unnormalized) distribution. *dist* is a $\{value : weight\}$ dictionary, where $weight \geq 0$. This returns a value with probability in proportion to its weight.

```

_____probStochSim.py — Probabilistic inference using stochastic simulation _____
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 multiple samples, and the distribution is over m values, it searches through the m values of the distribution for each sample.

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

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

```

_____probStochSim.py — (continued) _____
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 9.2

What is the time and space complexity of 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 a 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).

```

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

```

9.9.3 Rejection Sampling

```

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

```



```

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

```

9.9.4 Likelihood Weighting

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

```

                                     probStochSim.py — (continued)
104 class LikelihoodWeighting(SamplingInferenceMethod):
105     """The class that queries Graphical Models using Likelihood weighting.
106
107     gm is a belief network to query
108     """
109     method_name = "likelihood weighting"
110
111     def __init__(self, gm=None):
112         SamplingInferenceMethod.__init__(self, gm)
113
114     def query(self, qvar, obs={}, number_samples=1000, sample_order=None):
115         """computes P(qvar | obs) where
116         qvar is a variable.
117         obs is a {variable:value} dictionary.
118         sample_order is a list of factors where factors defining the parents
119         come before the factors for the child.
120         """
121         if sample_order is None:
122             sample_order = self.gm.topological_sort()
123         self.display(2, *[v for v in sample_order
124                           if v not in obs], sep="\t")
125         counts = {val:0 for val in qvar.domain}
126         for i in range(number_samples):
127             sample = {}
128             weight = 1.0
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 9.3 Change this algorithm so that it does **importance sampling** using a proposal distribution that may be different from the prior. It needs *sample_one* using a different distribution and then upadjustdate the weight of the current sample. For testing, use a proposal distribution that only differs from the prior for a subset of the variables. For which variables does the different poposal distribution make the most difference?

9.9.5 Particle Filtering

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

```

probStochSim.py — (continued)
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])]

```

```

172         for part in particles]
173     particles = [{**p, nvar:obs[nvar]}
174                 for p in resample(particles, weights,
175                                 number_samples)]
176
177     else:
178         for part in particles:
179             part[nvar] = sample_one({v:fac.get_value(**part,
180                                     nvar:v)})
181                                     for v in nvar.domain})
182     self.display(2,part[nvar],end="\t")
183     counts = {val:0 for val in qvar.domain}
184     for part in particles:
185         counts[part[qvar]] += 1
186     tot = sum(counts.values())
187     # as well as the distribution we also include the raw counts
188     dist = {c:v/tot for (c,v) in counts.items()}
189     dist["raw_counts"] = counts
190     return dist

```

Resampling

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

```

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

```

9.9.6 Examples

```

207 from probGraphicalModels import bn_4ch, A,B,C,D
208 bn_4chr = RejectionSampling(bn_4ch)
209 bn_4chL = LikelihoodWeighting(bn_4ch)
210 ## InferenceMethod.max_display_level = 2 # detailed tracing for all
    inference methods
211 ## bn_4chr.query(A,{})
212 ## bn_4chr.query(C,{})
213 ## bn_4chr.query(A,{C:True})
214 ## bn_4chr.query(B,{A:True,C:False})
215
216 from probExamples import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
217 bn_reportr = RejectionSampling(bn_report) # answers queries using
    rejection sampling
218 bn_reportL = LikelihoodWeighting(bn_report) # answers queries using
    likelihood weighting
219 bn_reportp = ParticleFiltering(bn_report) # answers queries using particle
    filtering
220 ## bn_reportr.query(Tamper,{})
221 ## bn_reportr.query(Tamper,{})
222 ## bn_reportr.query(Tamper,{Report:True})
223 ## InferenceMethod.max_display_level = 0 # no detailed tracing for all
    inference methods
224 ## bn_reportr.query(Tamper,{Report:True},number_samples=100000)
225 ## bn_reportr.query(Tamper,{Report:True,Smoke:False})
226 ## bn_reportr.query(Tamper,{Report:True,Smoke:False},number_samples=100)
227
228 ## bn_reportL.query(Tamper,{Report:True,Smoke:False},number_samples=100)
229 ## bn_reportL.query(Tamper,{Report:True,Smoke:False},number_samples=100)
230
231 from probExamples import bn_sprinkler,Season, Sprinkler
232 from probExamples import Rained, Grass_wet, Grass_shiny, Shoes_wet
233 bn_sprinklerr = RejectionSampling(bn_sprinkler) # answers queries using
    rejection sampling
234 bn_sprinklerL = LikelihoodWeighting(bn_sprinkler) # answers queries using
    rejection sampling
235 bn_sprinklerp = ParticleFiltering(bn_sprinkler) # answers queries using
    particle filtering
236 #bn_sprinklerr.query(Shoes_wet,{Grass_shiny:True,Rained:True})
237 #bn_sprinklerL.query(Shoes_wet,{Grass_shiny:True,Rained:True})
238 #bn_sprinklerp.query(Shoes_wet,{Grass_shiny:True,Rained:True})
239
240 if __name__ == "__main__":
241     InferenceMethod.testIM(RejectionSampling, threshold=0.1)
242     InferenceMethod.testIM(LikelihoodWeighting, threshold=0.1)
243     InferenceMethod.testIM(ParticleFiltering, threshold=0.1)

```

9.9.7 Gibbs Sampling

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

probStochSim.py — (continued)

```

245 #import random
246 #from probGraphicalModels import InferenceMethod
247
248 #from probStochSim import sample_one, SamplingInferenceMethod
249
250 class GibbsSampling(SamplingInferenceMethod):
251     """The class that queries Graphical Models using Gibbs Sampling.
252
253     bn is a graphical model (e.g., a belief network) to query
254     """
255     method_name = "Gibbs sampling"
256
257     def __init__(self, gm=None):
258         SamplingInferenceMethod.__init__(self, gm)
259         self.gm = gm
260
261     def query(self, qvar, obs={}, number_samples=1000, burn_in=100,
262              sample_order=None):
263         """computes P(qvar | obs) where
264         qvar is a variable.
265         obs is a {variable:value} dictionary.
266         sample_order is a list of non-observed variables in order, or
267         if sample_order None, an arbitrary ordering is used
268         """
269         counts = {val:0 for val in qvar.domain}
270         if sample_order is not None:
271             variables = sample_order
272         else:
273             variables = [v for v in self.gm.variables if v not in obs]
274             random.shuffle(variables)
275         var_to_factors = {v:set() for v in self.gm.variables}
276         for fac in self.gm.factors:
277             for var in fac.variables:
278                 var_to_factors[var].add(fac)
279         sample = {var:random.choice(var.domain) for var in variables}
280         self.display(3,"Sample:",sample)
281         sample.update(obs)
282         for i in range(burn_in + number_samples):
283             for var in variables:
284                 # get unnormalized probability distribution of var given its
285                 # neighbors
286                 vardist = {val:1 for val in var.domain}
287                 for val in var.domain:
288                     sample[var] = val
289                     for fac in var_to_factors[var]: # Markov blanket
290                         vardist[val] *= fac.get_value(sample)
291                 sample[var] = sample_one(vardist)
292             if i >= burn_in:
293                 counts[sample[qvar]] +=1

```

```

292         self.display(3, "      ", sample)
293         tot = sum(counts.values())
294         # as well as the computed distribution, we also include raw counts
295         dist = {c:v/tot for (c,v) in counts.items()}
296         dist["raw_counts"] = counts
297         self.display(2, f"Gibbs sampling P({qvar}|{obs}) = {dist}")
298         return dist
299
300 #from probGraphicalModels import bn_4ch, A,B,C,D
301 bn_4chg = GibbsSampling(bn_4ch)
302 ## InferenceMethod.max_display_level = 2 # detailed tracing for all
303     inference methods
304 bn_4chg.query(A,{})
305 ## bn_4chg.query(D,{})
306 ## bn_4chg.query(B,{D:True})
307 ## bn_4chg.query(B,{A:True,C:False})
308
309 from probExamples import bn_report,Alarm,Fire,Leaving,Report,Smoke,Tamper
310 bn_reportg = GibbsSampling(bn_report)
311 ## bn_reportg.query(Tamper,{Report:True},number_samples=1000)
312
313 if __name__ == "__main__":
314     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 Behavior of Stochastic Simulators

The stochastic simulation runs can give different answers each time they are run. For the algorithms that give the same answer in the limit as the number of samples approaches infinity (as do all of these algorithms), the algorithms can be compared by comparing the accuracy for multiple runs. Summary statistics like the variance may provide some information, but the assumptions behind the variance being appropriate (namely that the distribution is approximately 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

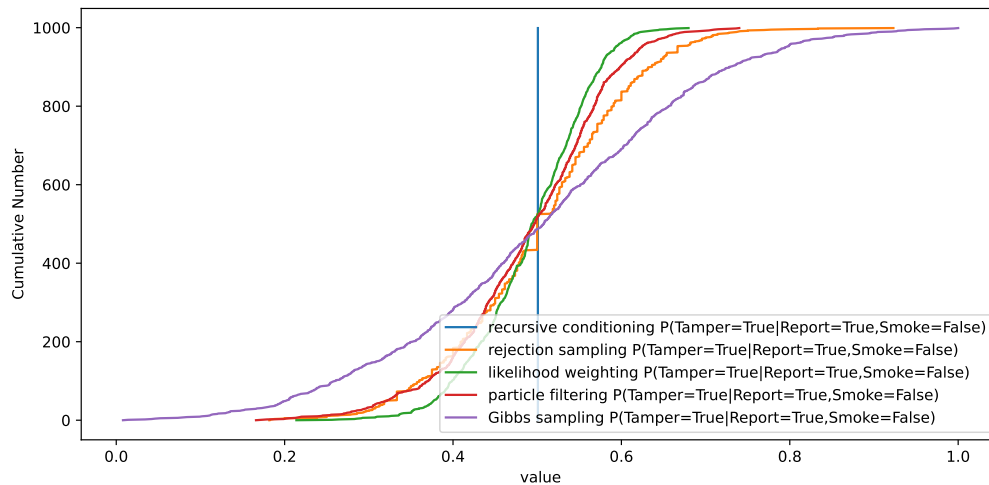


Figure 9.7: Cumulative distribution of the prediction of various models for $P(\text{Tamper}=\text{True} \mid \text{report} \wedge \neg \text{smoke})$

with prediction less than or equal to the x value. Thus this is like a cumulative distribution over the predictions, but with counts on the y -axis.

Note that for runs where there are no samples that are consistent with the observations (as can happen with rejection sampling), the prediction of probability is 1.0 (as a convention for $0/0$).

That variable *what* contains the query variable, or if *what* is “*prob_ev*”, the probability of evidence.

Figure 9.7 shows the distribution of various models. This figure is generated using the first `plot_mult` example below. Recursive conditioning gives the exact answer, and so is a vertical line. The others provide the cumulative prediction for 1000 runs for each method. This graph shows that for this graph and query, likelihood weighting is closest to the exact answer.

```

315 import matplotlib.pyplot as plt
316
317 def plot_stats(method, qvar, qval, obs, number_runs=1000, **queryargs):
318     """Plots a cumulative distribution of the prediction of the model.
319     method is a InferenceMethod (that implements appropriate query(.))
320     plots P(qvar=qval | obs)
321     qvar is the query variable, qval is corresponding value
322     obs is the {variable:value} dictionary representing the observations
323     number_runs is the number of runs that are plotted
324     **queryargs is the arguments to query (often number_samples for
325         sampling methods)
326     """
327     plt.ion()
328     plt.xlabel("value")
329     plt.ylabel("Cumulative Number")

```

```

329     method.max_display_level, prev_mdl = 0, method.max_display_level #no
        display
330     answers = [method.query(qvar, obs, **queryargs)
331                 for i in range(number_runs)]
332     values = [ans[qval] for ans in answers]
333     label = f""#{method.method_name}
        P({qvar}={qval})|{'', '.join(f'{var}={val}'
334                                     for (var, val) in
                                        obs.items())}""
335     values.sort()
336     plt.plot(values, range(number_runs), label=label)
337     plt.legend() #loc="upper left")
338     plt.draw()
339     method.max_display_level = prev_mdl # restore display level
340
341 # Try:
342 # plot_stats(bn_reportr, Tamper, True, {Report: True, Smoke: True},
        number_samples=1000, number_runs=1000)
343 # plot_stats(bn_reportL, Tamper, True, {Report: True, Smoke: True},
        number_samples=1000, number_runs=1000)
344 # plot_stats(bn_reportp, Tamper, True, {Report: True, Smoke: True},
        number_samples=1000, number_runs=1000)
345 # plot_stats(bn_reportr, Tamper, True, {Report: True, Smoke: True},
        number_samples=100, number_runs=1000)
346 # plot_stats(bn_reportL, Tamper, True, {Report: True, Smoke: True},
        number_samples=100, number_runs=1000)
347 # plot_stats(bn_reportg, Tamper, True, {Report: True, Smoke: True},
        number_samples=1000, number_runs=1000)
348
349 def plot_mult(methods, example, qvar, qval, obs, number_samples=1000,
        number_runs=1000):
350     for method in methods:
351         solver = method(example)
352         if isinstance(method, SamplingInferenceMethod):
353             plot_stats(solver, qvar, qval, obs,
                number_samples=number_samples, number_runs=number_runs)
354         else:
355             plot_stats(solver, qvar, qval, obs, number_runs=number_runs)
356
357 from probRC import ProbRC
358 # Try following (but it takes a while..)
359 methods = [ProbRC, RejectionSampling, LikelihoodWeighting,
        ParticleFiltering, GibbsSampling]
360 #plot_mult(methods, bn_report, Tamper, True, {Report: True, Smoke: False},
        number_samples=100, number_runs=1000)
361 # plot_mult(methods, bn_report, Tamper, True, {Report: False, Smoke: True},
        number_samples=100, number_runs=1000)
362
363 # Sprinkler Example:
364 # plot_stats(bn_sprinklerr, Shoes_wet, True, {Grass_shiny: True, Rained: True},

```



```

365     number_samples=1000)
# plot_stats(bn_sprinklerL, Shoes_wet, True, {Grass_shiny: True, Rained: True},
    number_samples=1000)

```

9.10 Hidden Markov Models

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

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

```

_____probHMM.py — Hidden Markov Model_____
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
65         sequence of
66         observations in obsseq using variable elimination.
67
68         Note that it first advances time.
69         This is what is required if it is called sequentially.
70         If that is not what is wanted initially, do an observe first.
71         """
72         for obs in obsseq:
73             self.advance() # advance time
74             self.observe(obs) # observe
75         return self.state_dist
76
77     def observe(self, obs):
78         """updates state conditioned on observations.
79         obs is a list of values for each observation variable"""
80         for i in self.hmm.obsvars:
81             self.state_dist = {st:self.state_dist[st]*(self.hmm.pobs[i][st]
82                                                         if obs[i] else
83                                                         (1-self.hmm.pobs[i][st]))
84                               for st in self.hmm.states}
85         norm = sum(self.state_dist.values()) # normalizing constant
86         self.state_dist = {st:self.state_dist[st]/norm for st in
87                             self.hmm.states}
88         self.display(2, "After observing", obs, "state
89                     distribution:", self.state_dist)
90
91     def advance(self):
92         """advance to the next time"""
93         nextstate = {st:0.0 for st in self.hmm.states} # distribution over
94                     next states
95         for j in self.hmm.states: # j ranges over next states
96             for i in self.hmm.states: # i ranges over previous states
97                 nextstate[j] += self.hmm.trans[i][j]*self.state_dist[i]
98         self.state_dist = nextstate
99         self.display(2, "After advancing state
100                    distribution:", self.state_dist)

```

The following are some queries for *hmm1*.

```

96 hmm1f1 = HMMVEfilter(hmm1)
97 # hmm1f1.filter([{'m1':0, 'm2':1, 'm3':1}, {'m1':1, 'm2':0, 'm3':1}])

```

```

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

```

Exercise 9.6 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.

```

_____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
        action.
        Instead of the transition probability, it has a function act2trans
        from action to transition probability.
        Any algorithms need to select the transition probability according
        to the action.
18     """
19
20     def __init__(self, states, obsvars, pobs, act2trans, indist):
21         self.act2trans = act2trans
22         HMM.__init__(self, states, obsvars, pobs, None, indist)
23
24
25
26

```

```

27 local_states = list(range(16))
28 door_positions = {2,4,7,11}
29 def prob_door(loc): return 0.8 if loc in door_positions else 0.1
30 local_obs = {'door':[prob_door(i) for i in range(16)]}
31 act2trans = {'right': [[0.1 if next == current
32                        else 0.8 if next == (current+1)%16
33                        else 0.074 if next == (current+2)%16
34                        else 0.002 for next in range(16)]
35                  for current in range(16)],
36              'left': [[0.1 if next == current
37                      else 0.8 if next == (current-1)%16
38                      else 0.074 if next == (current-2)%16
39                      else 0.002 for next in range(16)]
40                      for current in range(16)]}
41 hmm_16pos = HMM_Controlled(local_states, {'door'}, local_obs,
42                             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)_____
43 class HMM_Local(HMMVEfilter):
44     """VE filter for controlled HMMs
45     """
46     def __init__(self, hmm):
47         HMMVEfilter.__init__(self, hmm)
48
49     def go(self, action):
50         self.hmm.trans = self.hmm.act2trans[action]
51         self.advance()
52
53 loc_filt = HMM_Local(hmm_16pos)
54 # loc_filt.observe({'door':True}); loc_filt.go("right");
55   loc_filt.observe({'door':False}); loc_filt.go("right");
56   loc_filt.observe({'door':True})
57 # loc_filt.state_dist

```

The following lets us interactively move the agent and provide observations. It shows the distribution over locations. Figure 9.8 shows the GUI obtained by `Show_Localization(hmm_16pos)` after some interaction.

```

_____probLocalization.py — (continued)_____
57 class Show_Localization(Displayable):
58     def __init__(self, hmm, fontsize=10):
59         self.hmm = hmm
60         self.fontsize = fontsize
61         self.loc_filt = HMM_Local(hmm)
62         fig,(self.ax) = plt.subplots()
63         plt.subplots_adjust(bottom=0.2)
64         ## Set up buttons:
65         left_but = Button(plt.axes([0.05,0.02,0.1,0.05]), "left")

```

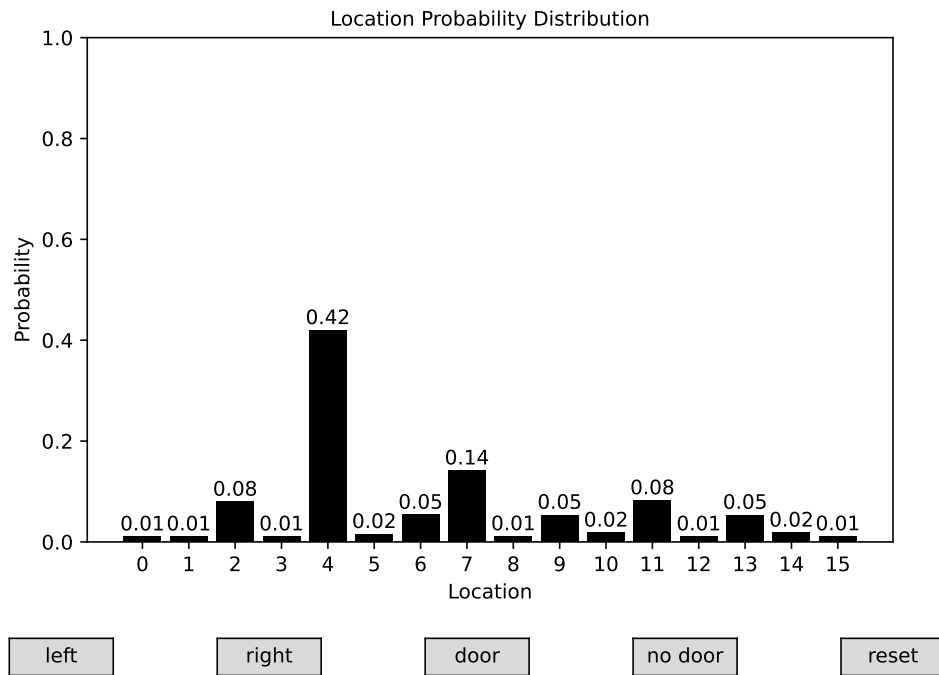


Figure 9.8: Localization GUI after observing a door, moving right, observing no door, moving right, and observing a door.

```

66     left_buttn.label.set_fontsize(self.fontsize)
67     left_buttn.on_clicked(self.left)
68     right_buttn = Button(plt.axes([0.25,0.02,0.1,0.05]), "right")
69     right_buttn.label.set_fontsize(self.fontsize)
70     right_buttn.on_clicked(self.right)
71     door_buttn = Button(plt.axes([0.45,0.02,0.1,0.05]), "door")
72     door_buttn.label.set_fontsize(self.fontsize)
73     door_buttn.on_clicked(self.door)
74     nodoor_buttn = Button(plt.axes([0.65,0.02,0.1,0.05]), "no door")
75     nodoor_buttn.label.set_fontsize(self.fontsize)
76     nodoor_buttn.on_clicked(self.nodoor)
77     reset_buttn = Button(plt.axes([0.85,0.02,0.1,0.05]), "reset")
78     reset_buttn.label.set_fontsize(self.fontsize)
79     reset_buttn.on_clicked(self.reset)
80     ## draw the distribution
81     plt.subplot(1, 1, 1)
82     self.draw_dist()
83     plt.show()
84
85     def draw_dist(self):
86         self.ax.clear()
87         plt.ylim(0,1)

```

```

88     plt.ylabel("Probability", fontsize=self.fontsize)
89     plt.xlabel("Location", fontsize=self.fontsize)
90     plt.title("Location Probability Distribution",
91             fontsize=self.fontsize)
92     plt.xticks(self.hmm.states, fontsize=self.fontsize)
93     plt.yticks(fontsize=self.fontsize)
94     vals = [self.loc_filt.state_dist[i] for i in self.hmm.states]
95     self.bars = self.ax.bar(self.hmm.states, vals, color='black')
96     self.ax.bar_label(self.bars, ["{v:.2f}".format(v=v) for v in vals],
97                        padding = 1, fontsize=self.fontsize)
98     plt.draw()
99
100     def left(self, event):
101         self.loc_filt.go("left")
102         self.draw_dist()
103     def right(self, event):
104         self.loc_filt.go("right")
105         self.draw_dist()
106     def door(self, event):
107         self.loc_filt.observe({'door': True})
108         self.draw_dist()
109     def nodoor(self, event):
110         self.loc_filt.observe({'door': False})
111         self.draw_dist()
112     def reset(self, event):
113         self.loc_filt.state_dist = {i: 1/16 for i in range(16)}
114         self.draw_dist()
115 # Show_Localization(hmm_16pos)
116 # Show_Localization(hmm_16pos, fontsize=15) # for demos - enlarge window

```

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 (or use Python's random library).

9.10.4 Generating Examples

The following code is useful for generating examples.

```

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])

```

```

196     return stateseq, obsseq
197
198 def simobs(hmm, stateseq):
199     """returns observation sequence for the state sequence"""
200     obsseq = []
201     for state in stateseq:
202         newobs =
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, consider an arbitrary point, *now*, which will be represented as time 1. Each variable will have a corresponding previous variable; the variables and their previous instances will be created together.

A dynamic belief network consists of:

- A set of features. A variable is a feature-time pair.
- An initial distribution over the features “now” (time 1). This is a belief network with all variables being time 1 variables.
- A specification of the dynamics. We define the how the variables *now* (time 1) depend on variables *now* and the previous time (time 0), in such a way that the graph is acyclic.

```

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 DBN variable has both a name and an index. The index defaults to 1.
21     position is (x,y) where x>0.3
22     """
23     def __init__(self, name, domain=[False,True], index=1, position=None):
24         Variable.__init__(self, f"{name}_{index}", domain,
25                             position=position)
26         self.basename = name
27         self.domain = domain
28         self.index = index
29         self.previous = None
30
31     def __lt__(self, other):
32         if self.name == other.name:
33             return self.index < other.index
34         else:
35             return self.name < other.name
36
37 def variable_pair(name, domain=[False,True], position=None):
38     """returns a variable and its predecessor. This is used to define
39     2-stage DBNs
40
41     If the name is X, it returns the pair of variables X_prev,X_now"""
42     var_now = DBNvariable(name, domain, index='now', position=position)
43     if position:
44         (x,y) = position
45         position = (x-0.3, y)
46         var_prev = DBNvariable(name, domain, index='prev', position=position)
47         var_now.previous = var_prev
48     return var_prev, var_now

```

A *FactorRename* is a factor that is the result of renaming the variables in the factor. It takes a factor, *fac*, and a $\{new : old\}$ dictionary, where *new* is the name

of a variable in the resulting factor and *old* is the corresponding name in *fac*. This assumes that all variables are renamed.

```

_____probDBN.py — (continued) _____
48 class FactorRename(Factor):
49     def __init__(self, fac, renaming):
50         """A renamed factor.
51         fac is a factor
52         renaming is a dictionary of the form {new:old} where old and new
53         var variables,
54         where the variables in fac appear exactly once in the renaming
55         """
56         Factor.__init__(self, [n for (n,o) in renaming.items() if o in
57                               fac.variables])
58         self.orig_fac = fac
59         self.renaming = renaming
60
61     def get_value(self, assignment):
62         return self.orig_fac.get_value({self.renaming[var]:val
63                                         for (var,val) in assignment.items()
64                                         if var in self.variables})

```

The following class renames the variables of a conditional probability distribution. It is used for template models (e.g., dynamic decision networks or relational models)

```

_____probDBN.py — (continued) _____
64 class CPDrename(FactorRename, CPD):
65     def __init__(self, cpd, renaming):
66         renaming_inverse = {old:new for (new,old) in renaming.items()}
67         CPD.__init__(self, renaming_inverse[cpd.child], [renaming_inverse[p]
68                                                         for p in cpd.parents])
69         self.orig_fac = cpd
70         self.renaming = renaming

```

```

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

```

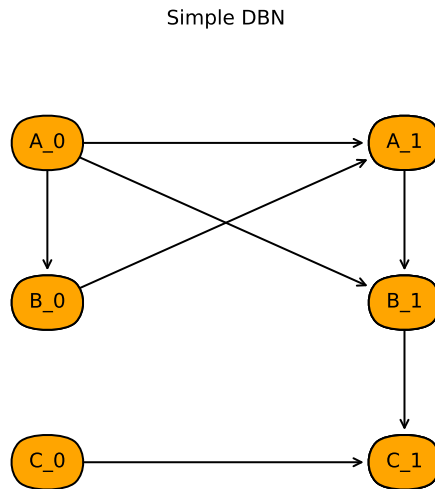


Figure 9.9: Simple dynamic belief network (dbn1.show())

```

84     self.title = title
85     self.vars_now = vars_now
86     self.vars_prev = [v.previous for v in vars_now]
87     self.transition_factors = transition_factors
88     self.init_factors = init_factors
89     self.var_index = {} # var_index[v] is the index of variable v
90     for i,v in enumerate(vars_now):
91         self.var_index[v]=i
92
93     def show(self):
94         BNfromDBN(self,1).show()

```

Here is a 3 variable DBN (shown in Figure 9.9):

```

probDBN.py — (continued)
96 A0,A1 = variable_pair("A", domain=[False,True], position = (0.4,0.8))
97 B0,B1 = variable_pair("B", domain=[False,True], position = (0.4,0.5))
98 C0,C1 = variable_pair("C", domain=[False,True], position = (0.4,0.2))
99
100 # dynamics
101 pc = Prob(C1,[B1,C0],[[0.03,0.97],[0.38,0.62]],[[0.23,0.77],[0.78,0.22]])
102 pb = Prob(B1,[A0,A1],[[0.5,0.5],[0.77,0.23]],[[0.4,0.6],[0.83,0.17]])
103 pa = Prob(A1,[A0,B0],[[0.1,0.9],[0.65,0.35]],[[0.3,0.7],[0.8,0.2]])
104
105 # initial distribution
106 pa0 = Prob(A1,[],[0.9,0.1])
107 pb0 = Prob(B1,[A1],[[0.3,0.7],[0.8,0.2]])
108 pc0 = Prob(C1,[],[0.2,0.8])
109
110 dbn1 = DBN("Simple DBN",[A1,B1,C1],[pa,pb,pc],[pa0,pb0,pc0])

```

Here is the animal example

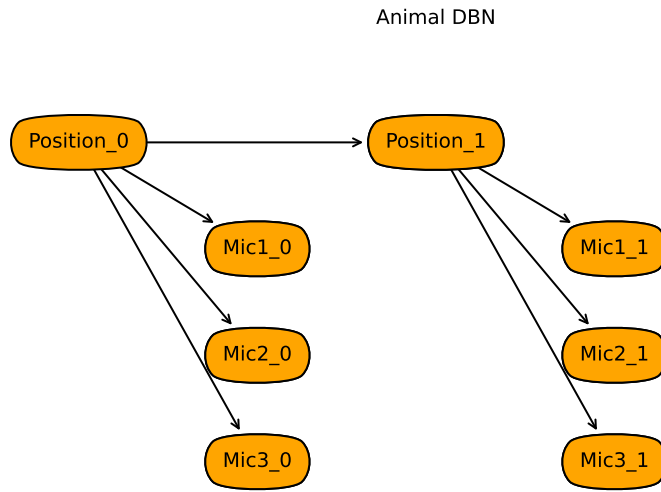


Figure 9.10: Animal dynamic belief network (dbn_an.show())

```

112 from probHMM import closeMic, farMic, midMic, sm, mmc, sc, mcm, mcc
113
114 Pos_0,Pos_1 = variable_pair("Position", domain=[0,1,2,3],
115                             position=(0.5,0.8))
115 Mic1_0,Mic1_1 = variable_pair("Mic1", position=(0.6,0.6))
116 Mic2_0,Mic2_1 = variable_pair("Mic2", position=(0.6,0.4))
117 Mic3_0,Mic3_1 = variable_pair("Mic3", position=(0.6,0.2))
118
119 # conditional probabilities - see hmm for the values of sm,mmc, etc
120 ppos = Prob(Pos_1, [Pos_0],
121             [[sm, mmc, mmc, mmc], #was in middle
122              [mcm, sc, mcc, mcc], #was in corner 1
123              [mcm, mcc, sc, mcc], #was in corner 2
124              [mcm, mcc, mcc, sc]]) #was in corner 3
125 pm1 = Prob(Mic1_1, [Pos_1], [[1-midMic, midMic], [1-closeMic, closeMic],
126                             [1-farMic, farMic], [1-farMic, farMic]])
127 pm2 = Prob(Mic2_1, [Pos_1], [[1-midMic, midMic], [1-farMic, farMic],
128                             [1-closeMic, closeMic], [1-farMic, farMic]])
129 pm3 = Prob(Mic3_1, [Pos_1], [[1-midMic, midMic], [1-farMic, farMic],
130                             [1-farMic, farMic], [1-closeMic, closeMic]])
131 ipos = Prob(Pos_1,[], [0.25, 0.25, 0.25, 0.25])
132 dbn_an =DBN("Animal DBN",[Pos_1,Mic1_1,Mic2_1,Mic3_1],
133             [ppos, pm1, pm2, pm3],

```

```
134 |         [ipos, pm1, pm2, pm3])
```

9.11.2 Unrolling DBNs

```

136 |         probDBN.py — (continued)
137 |
138 |
139 |
140 | class BNfromDBN(BeliefNetwork):
141 |     """Belief Network unrolled from a dynamic belief network
142 |     """
143 |
144 |     def __init__(self,dbn,horizon):
145 |         """dbn is the dynamic belief network being unrolled
146 |         horizon>0 is the number of steps (so there will be horizon+1
147 |         variables for each DBN variable.
148 |         """
149 |         self.dbn = dbn
150 |         self.horizon = horizon
151 |         self.minx,self.width = None, None # for positions pf variables
152 |         self.name2var = {var.basename:
153 |             [DBNvariable(var.basename,var.domain,index,
154 |                 position=self.pos(var,index))
155 |                 for index in range(horizon+1)]
156 |                 for var in dbn.vars_now}
157 |         self.display(1,f"name2var={self.name2var}")
158 |         variables = {v for vs in self.name2var.values() for v in vs}
159 |         self.display(1,f"variables={variables}")
160 |         bnfactors = {CPDrename(fac,{self.name2var[var.basename][0]:var
161 |             for var in fac.variables})
162 |             for fac in dbn.init_factors}
163 |         bnfactors |= {CPDrename(fac,{self.name2var[var.basename][i]:var
164 |             for var in fac.variables if
165 |                 var.index=='prev'}
166 |                 | {self.name2var[var.basename][i+1]:var
167 |                 for var in fac.variables if
168 |                     var.index=='now'})
169 |             for fac in dbn.transition_factors
170 |             for i in range(horizon)}
171 |         self.display(1,f"bnfactors={bnfactors}")
172 |         BeliefNetwork.__init__(self, dbn.title, variables, bnfactors)
173 |
174 |     def pos(self, var, index):
175 |         minx = min(x for (x,y) in (var.position for var in
176 |             self.dbn.vars_now))-1e-6
177 |         maxx = max(x for (x,y) in (var.position for var in
178 |             self.dbn.vars_now))
179 |         width = maxx-minx
180 |         xo,yo = var.position
181 |         xi = index/(self.horizon+1)+(xo-minx)/width/(self.horizon+1)/2
182 |         return (xi, yo)

```

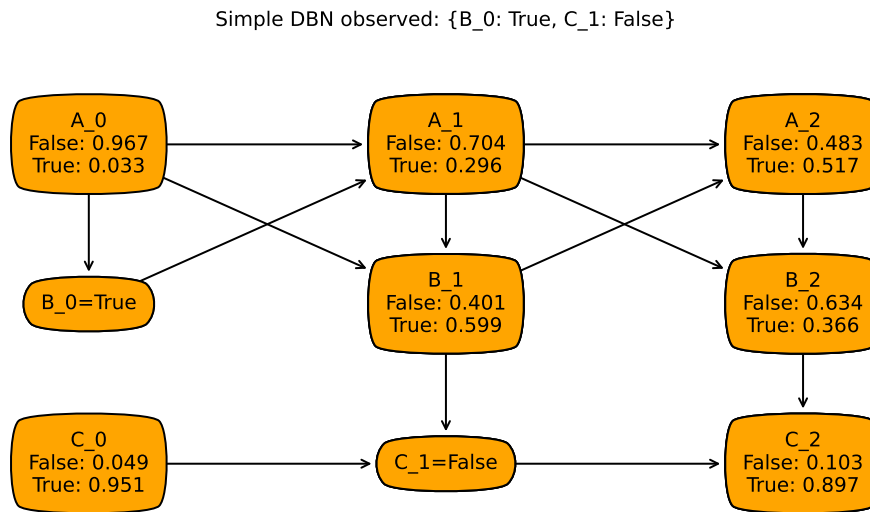


Figure 9.11: Simple dynamic belief network (dbn1) horizon 2

Here are two examples. You use `bn.name2var['B'][2]` to get the variable B2 (B at time 2). Figure 9.11 shows the output of the `drc.show_post` below:

```

174 # Try
175 from probRC import ProbRC
176 # bn = BNfromDBN(dbn1,2) # construct belief network
177 # drc = ProbRC(bn)         # initialize recursive conditioning
178 # B2 = bn.name2var['B'][2]
179 # drc.query(B2) #P(B2)
180 #
181     drc.query(bn.name2var['B'][1],{bn.name2var['B'][0]:True,bn.name2var['C'][1]:False})
182     #P(B1|b0,~c1)
183 # drc.show_post({bn.name2var['B'][0]:True,bn.name2var['C'][1]:False})
184
185 # Plot Distributions:
186 # bna = BNfromDBN(dbn_an,5) # animal belief network with horizon 5
187 # dra = ProbRC(bna)
188 # dra.show_post(obs =
189     {bna.name2var['Mic1'][1]:True,bna.name2var['Mic1'][2]:True})

```

9.11.3 DBN Filtering

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

probDBN.py — (continued)

```

188 class DBNVEfilter(VE):
189     def __init__(self, dbn):
190         self.dbn = dbn
191         self.current_factors = dbn.init_factors
192         self.current_obs = {}
193
194     def observe(self, obs):
195         """updates the current observations with obs.
196         obs is a variable:value dictionary where variable is a current
197         variable.
198         """
199         assert all(self.current_obs[var]==obs[var] for var in obs
200                   if var in self.current_obs), "inconsistent current
201                   observations"
202         self.current_obs.update(obs) # note 'update' is a dict method
203
204     def query(self, var):
205         """returns the posterior probability of current variable var"""
206         return
207             VE(GraphicalModel(self.dbn.title, self.dbn.vars_now, self.current_factors)
208               ).query(var, self.current_obs)
209
210     def advance(self):
211         """advance to the next time"""
212         prev_factors = [self.make_previous(fac) for fac in
213                       self.current_factors]
214         prev_obs = {var.previous:val for var, val in
215                   self.current_obs.items()}
216         two_stage_factors = prev_factors + self.dbn.transition_factors
217         self.current_factors =
218             self.elim_vars(two_stage_factors, self.dbn.vars_prev, prev_obs)
219         self.current_obs = {}
220
221     def make_previous(self, fac):
222         """Creates new factor from fac where the current variables in fac
223         are renamed to previous variables.
224         """
225         return FactorRename(fac, {var.previous:var for var in
226                               fac.variables})
227
228     def elim_vars(self, factors, vars, obs):
229         for var in vars:
230             if var in obs:
231                 factors = [self.project_observations(fac, obs) for fac in
232                           factors]
233             else:
234                 factors = self.eliminate_var(factors, var)
235         return factors

```

Example queries:

```
_____probDBN.py — (continued) _____
230 #df = DBNVEfilter(dbn1)
231 #df.observe({B1:True}); df.advance(); df.observe({C1:False})
232 #df.query(B1) #P(B1|B0,C1)
233 #df.advance(); df.query(B1)
234 #dfa = DBNVEfilter(dbn_an)
235 # dfa.observe({Mic1_1:0, Mic2_1:1, Mic3_1:1})
236 # dfa.advance()
237 # dfa.observe({Mic1_1:1, Mic2_1:0, Mic3_1:1})
238 # dfa.query(Pos_1)
```

Learning with Uncertainty

10.1 Bayesian Learning

The section contains two implementations of the (discretized) beta distribution. The first represents Bayesian learning as a belief network. The second is an interactive tool to understand the beta distribution.

The following uses a belief network representation from the previous chapter to learn (discretized) probabilities. Figure 10.1 shows the output after observing *heads, heads, tails*. Notice the prediction of future tosses.

```
learnBayesian.py — Bayesian Learning
11 from variable import Variable
12 from probFactors import Prob
13 from probGraphicalModels import BeliefNetwork
14 from probRC import ProbRC
15
16 ##### Coin Toss ###
17 # multiple coin tosses:
18 toss = ['tails','heads']
19 tosses = [ Variable(f"Toss#{i}", toss,
20                  (0.8, 0.9-i/10) if i<10 else (0.4,0.2))
21            for i in range(11)]
22
23 def coinTossBN(num_bins = 10):
24     prob_bins = [x/num_bins for x in range(num_bins+1)]
25     PH = Variable("P_heads", prob_bins, (0.1,0.9))
26     p_PH = Prob(PH,[],{x:0.5/num_bins if x in [0,1] else 1/num_bins for x
27                  in prob_bins})
28     p_tosses = [ Prob(tosses[i],[PH], {x:{'tails':1-x,'heads':x} for x in
29                  prob_bins})
30                 for i in range(11)]
```

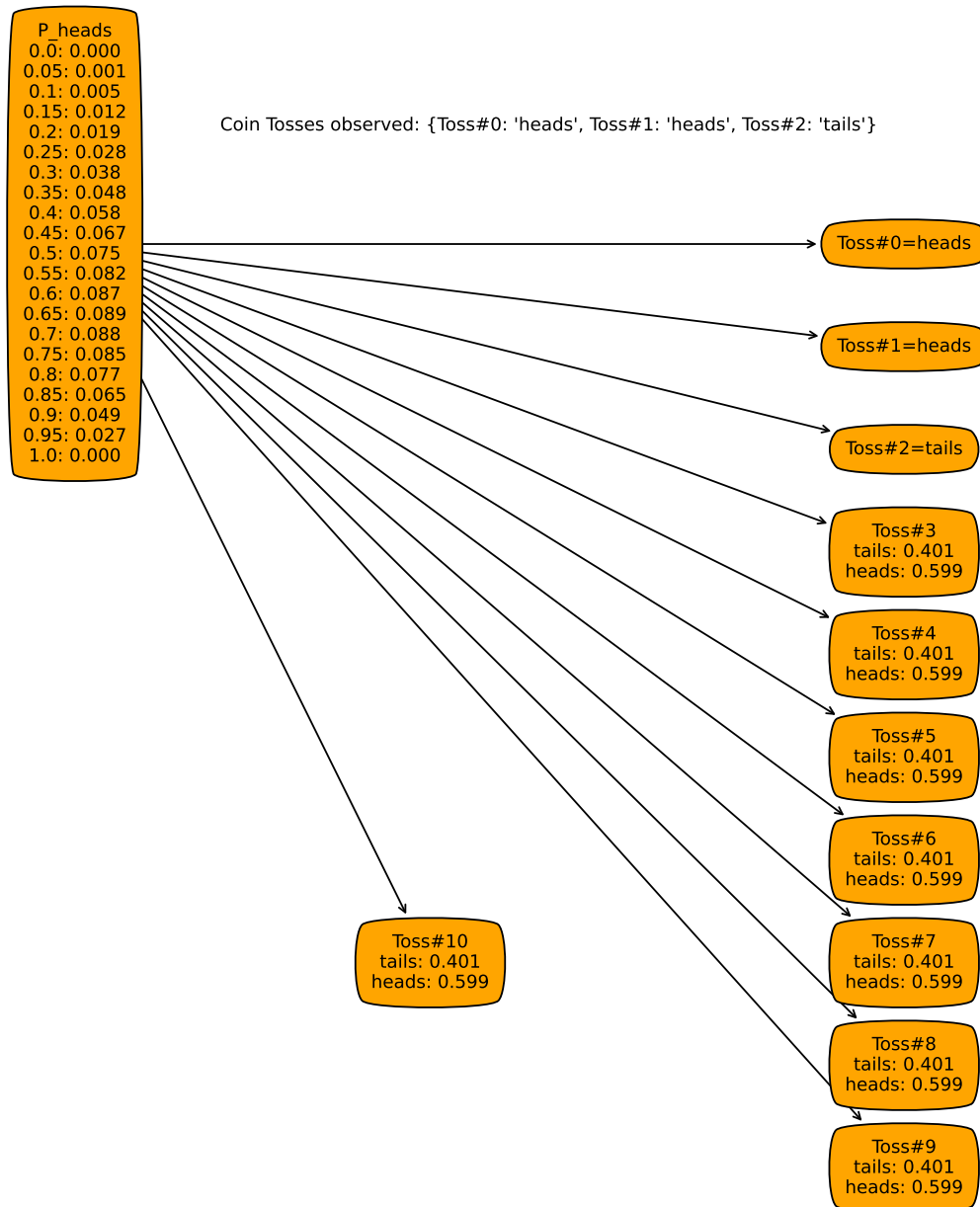


Figure 10.1: coinTossBN after observing heads, heads, tails

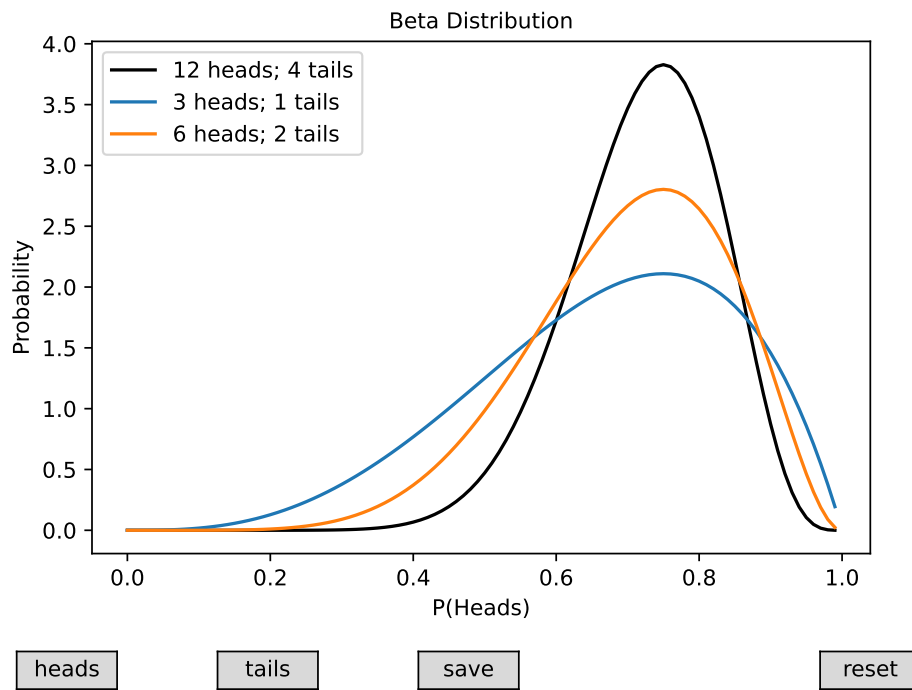


Figure 10.2: Beta distribution after some observations

```

29     return BeliefNetwork("Coin Tosses",
30                           [PH]+tosses,
31                           [p_PH]+p_tosses)
32
33
34 #
35 # coinRC = ProbRC(coinTossBN(20))
36 # coinRC.query(tosses[10],{tosses[0]: 'heads'})
37 # coinRC.show_post({})
38 # coinRC.show_post({tosses[0]: 'heads'})
39 # coinRC.show_post({tosses[0]: 'heads', tosses[1]: 'heads'})
40 # coinRC.show_post({tosses[0]: 'heads', tosses[1]: 'heads', tosses[2]: 'tails'})

```

Figure 10.2 shows a plot of the Beta distribution (the P_{head} variable in the previous belief network) given some sets of observations.

This is a plot that is produced by the following interactive tool.

```

learnBayesian.py — (continued)
42 from display import Displayable
43 import matplotlib.pyplot as plt
44 from matplotlib.widgets import Button, CheckButtons
45
46 class Show_Beta(Displayable):

```

```

47 def __init__(self,num=100, fontsize=10):
48     self.num = num
49     self.dist = [1 for i in range(num)]
50     self.vals = [i/num for i in range(num)]
51     self.fontsize = fontsize
52     self.saves = []
53     self.num_heads = 0
54     self.num_tails = 0
55     plt.ioff()
56     fig,(self.ax) = plt.subplots()
57     plt.subplots_adjust(bottom=0.2)
58     ## Set up buttons:
59     heads_but = Button(plt.axes([0.05,0.02,0.1,0.05]), "heads")
60     heads_but.label.set_fontsize(self.fontsize)
61     heads_but.on_clicked(self.heads)
62     tails_but = Button(plt.axes([0.25,0.02,0.1,0.05]), "tails")
63     tails_but.label.set_fontsize(self.fontsize)
64     tails_but.on_clicked(self.tails)
65     save_but = Button(plt.axes([0.45,0.02,0.1,0.05]), "save")
66     save_but.label.set_fontsize(self.fontsize)
67     save_but.on_clicked(self.save)
68     reset_but = Button(plt.axes([0.85,0.02,0.1,0.05]), "reset")
69     reset_but.label.set_fontsize(self.fontsize)
70     reset_but.on_clicked(self.reset)
71     ## draw the distribution
72     plt.subplot(1, 1, 1)
73     self.draw_dist()
74     plt.show()
75
76 def draw_dist(self):
77     sv = self.num/sum(self.dist)
78     self.dist = [v*sv for v in self.dist]
79     #print(self.dist)
80     self.ax.clear()
81     plt.ylabel("Probability", fontsize=self.fontsize)
82     plt.xlabel("P(Heads)", fontsize=self.fontsize)
83     plt.title("Beta Distribution", fontsize=self.fontsize)
84     plt.xticks(fontsize=self.fontsize)
85     plt.yticks(fontsize=self.fontsize)
86     self.ax.plot(self.vals, self.dist, color='black', label =
87         f"{self.num_heads} heads; {self.num_tails} tails")
88     for (nh,nt,d) in self.saves:
89         self.ax.plot(self.vals, d, label = f"{nh} heads; {nt} tails")
90     self.ax.legend()
91     plt.draw()
92
93 def heads(self,event):
94     self.num_heads += 1
95     self.dist = [self.dist[i]*self.vals[i] for i in range(self.num)]
96     self.draw_dist()

```

```

96     def tails(self,event):
97         self.num_tails += 1
98         self.dist = [self.dist[i]*(1-self.vals[i]) for i in range(self.num)]
99         self.draw_dist()
100    def save(self,event):
101        self.saves.append((self.num_heads,self.num_tails,self.dist))
102        self.draw_dist()
103    def reset(self,event):
104        self.num_tails = 0
105        self.num_heads = 0
106        self.dist = [1/self.num for i in range(self.num)]
107        self.draw_dist()
108
109    # s1 = Show_Beta(100)
110    # s1 = Show_Beta(100, fontsize=15) # for demos - enlarge window

```

10.2 K-means

The k-means learner takes in a dataset and a number of classes, and learns a mapping from examples to classes (*class_of_eg*) and a function that makes predictions for classes (*class_predictions*).

It maintains two lists that suffice as sufficient statistics to classify examples, and to learn the classification:

- *class_counts* is a list such that *class_counts*[*c*] is the number of examples in the training set with *class* = *c*.
- *feature_sum* is a list such that *feature_sum*[*f*][*c*] is sum of the values for the feature *f* for members of class *c*. The average value of the *i*th feature in class *i* is

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

when *class_counts*[*c*] > 0 and is 0 otherwise.

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

```

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

```

```

20     self.random_initialize()
21     self.max_display_level = 5
22
23     def random_initialize(self):
24         # class_counts[c] is the number of examples with class=c
25         self.class_counts = [0]*self.num_classes
26         # feature_sum[f][c] is the sum of the values of feature f for class
27         # c
28         self.feature_sum = {feat:[0]*self.num_classes
29                             for feat in self.dataset.input_features}
30         for eg in self.dataset.train:
31             cl = random.randrange(self.num_classes) # assign eg to random
32             class
33             self.class_counts[cl] += 1
34             for feat in self.dataset.input_features:
35                 self.feature_sum[feat][cl] += feat(eg)
36         self.num_iterations = 0
37         self.display(1,"Initial class counts: ",self.class_counts)

```

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

```

learnKMeans.py — (continued)
37     def distance(self,cl,eg):
38         """distance of the eg from the mean of the class"""
39         return sum( (self.class_prediction(feat,cl)-feat(eg))**2
40                     for feat in self.dataset.input_features)
41
42     def class_prediction(self,feat,cl):
43         """prediction of the class cl on the feature with index feat_ind"""
44         if self.class_counts[cl] == 0:
45             return 0 # arbitrary prediction
46         else:
47             return self.feature_sum[feat][cl]/self.class_counts[cl]
48
49     def class_of_eg(self,eg):
50         """class to which eg is assigned"""
51         return (min((self.distance(cl,eg),cl)
52                     for cl in range(self.num_classes)))[1]
53         # second element of tuple, which is a class with minimum
54         # distance

```

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

```

learnKMeans.py — (continued)
55     def k_means_step(self):

```



```

56     """Updates the model with one step of k-means.
57     Returns whether the assignment is stable.
58     """
59     new_class_counts = [0]*self.num_classes
60     # feature_sum[f][c] is the sum of the values of feature f for class
        c
61     new_feature_sum = {feat: [0]*self.num_classes
62                          for feat in self.dataset.input_features}
63     for eg in self.dataset.train:
64         cl = self.class_of_eg(eg)
65         new_class_counts[cl] += 1
66         for feat in self.dataset.input_features:
67             new_feature_sum[feat][cl] += feat(eg)
68     stable = (new_class_counts == self.class_counts) and
        (self.feature_sum == new_feature_sum)
69     self.class_counts = new_class_counts
70     self.feature_sum = new_feature_sum
71     self.num_iterations += 1
72     return stable
73
74
75     def learn(self,n=100):
76         """do n steps of k-means, or until convergence"""
77         i=0
78         stable = False
79         while i<n and not stable:
80             stable = self.k_means_step()
81             i += 1
82             self.display(1,"Iteration",self.num_iterations,
83                          "class counts: ",self.class_counts,"
84                          Stable=",stable)
85
86         return stable
87
88     def show_classes(self):
89         """sorts the data by the class and prints in order.
90         For visualizing small data sets
91         """
92         class_examples = [[] for i in range(self.num_classes)]
93         for eg in self.dataset.train:
94             class_examples[self.class_of_eg(eg)].append(eg)
95         print("Class","Example",sep='\t')
96         for cl in range(self.num_classes):
97             for eg in class_examples[cl]:
98                 print(cl,*eg,sep='\t')

```

Figure 10.3 shows multiple runs for Example 10.5 in Section 10.3.1 of Poole and Mackworth [2023]. Note that the y -axis is sum of squares of the values, which is the square of the Euclidian distance. K-means can stabilize on a different assignment each time it is run. The first run with 2 classes shown in the figure was stable after the first step. The next two runs with 3 classes started

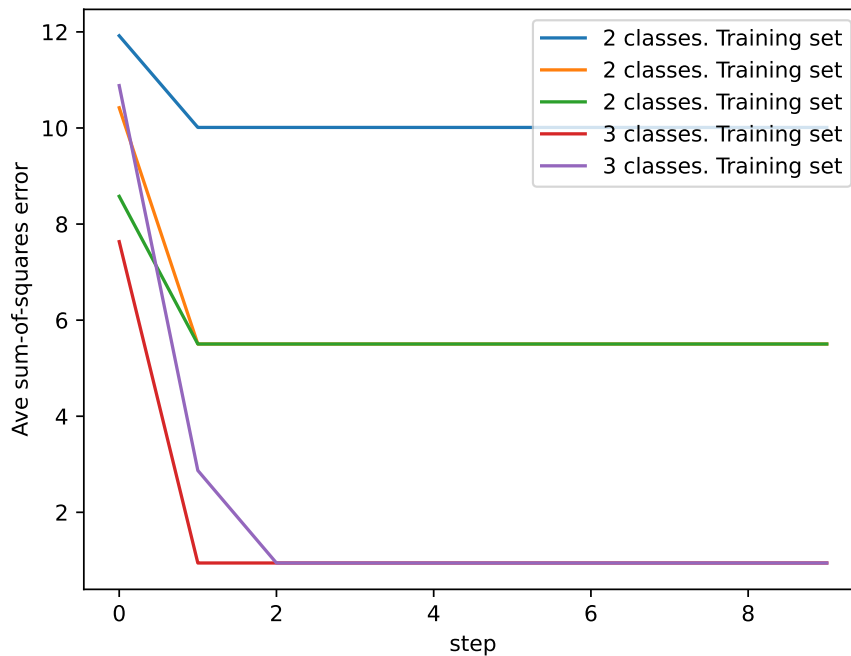


Figure 10.3: k-means plotting error.

with different assignments, but stabilized on the same assignment. (You cannot check if it is the same assignment from the graph, but need to check the assignment of examples to classes.) The second run with 3 classes took two steps to stabilize, but the other only took one. Note that the algorithm only determines that it is stable with one more run.

```

97         learnKMeans.py — (continued)
98     def plot_error(self, maxstep=20):
99         """Plots the sum-of-squares error as a function of the number of
100            steps"""
101         plt.ion()
102         plt.xlabel("step")
103         plt.ylabel("Ave sum-of-squares error")
104         train_errors = []
105         if self.dataset.test:
106             test_errors = []
107         for i in range(maxstep):
108             train_errors.append( sum(self.distance(self.class_of_eg(eg), eg)
109                                   for eg in self.dataset.train)
110                               /len(self.dataset.train))
111             if self.dataset.test:
112                 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     self.learn(1)
115     plt.plot(range(maxstep), train_errors,
116             label=str(self.num_classes)+" classes. Training set")
117     if self.dataset.test:
118         plt.plot(range(maxstep), test_errors,
119                 label=str(self.num_classes)+" classes. Test set")
120     plt.legend()
121     plt.draw()
122
123 # data = Data_from_file('data/emdata1.csv', num_train=10,
124 #                       target_index=2000) # trivial example
125 data = Data_from_file('data/emdata2.csv', num_train=10, target_index=2000)
126 # data = Data_from_file('data/emdata0.csv', num_train=14,
127 #                       target_index=2000) # example from textbook
128 kml = K_means_learner(data,2)
129 num_iter=4
130 print("Class assignment after",num_iter,"iterations:")
131 kml.learn(num_iter); kml.show_classes()
132
133 # Plot the error
134 # km2=K_means_learner(data,2); km2.plot_error(10) # 2 classes
135 # km3=K_means_learner(data,3); km3.plot_error(10) # 3 classes
136 # km13=K_means_learner(data,10); km13.plot_error(10) # 10 classes
137
138 # data = Data_from_file('data/carbool.csv', target_index=2000,
139 #                       one_hot=True)
140 # kml = K_means_learner(data,3)
141 # kml.learn(20); kml.show_classes()
142 # km3=K_means_learner(data,3); km3.plot_error(10) # 3 classes
143 # km3=K_means_learner(data,10); km3.plot_error(10) # 10 classes

```

Exercise 10.1 If there are many classes, some of the classes can become empty (e.g., try 100 classes with carbool.csv). Implement a way to put some examples into a class, if possible. Two ideas are:

- Initialize the classes with actual examples, so that the classes will not start empty. (Do the classes become empty?)
- In *class prediction*, we test whether the code is empty, and make a prediction of 0 for an empty class. It is possible to make a different prediction to “steal” an example (but you should make sure that a class has a consistent value for each feature in a loop).

Make your own suggestions, and compare it with the original, and whichever of these you think may work better.

10.3 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 through the training examples, and updates these counts. The first time it is run, when there is no model, it uses random distributions.

```

learnEM.py — (continued)
23 def em_step(self, orig_class_counts, orig_feature_counts):
24     """updates the model."""
25     class_counts = [0]*self.num_classes
26     feature_counts = [{val:[0]*self.num_classes
27                         for val in feat.frange}
28                      for feat in self.dataset.input_features]
29     for tple in self.dataset.train:
30         if orig_class_counts: # a model exists
31             tpl_class_dist = self.prob(tple, orig_class_counts,
32                                       orig_feature_counts)

```

```

32         else:                                # initially, with no model, return a random
33             distribution
34             tpl_class_dist = random_dist(self.num_classes)
35         for cl in range(self.num_classes):
36             class_counts[cl] += tpl_class_dist[cl]
37             for (ind, feat) in enumerate(self.dataset.input_features):
38                 feature_counts[ind][feat(tple)][cl] += tpl_class_dist[cl]
39     return class_counts, feature_counts

```

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

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

The last step is because $\text{len}(\text{self.dataset})$ is a constant (independent of *c*). $\text{class_counts}[c]$ 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     thesum = sum(unnorm)
49     return [un/thesum for un in unnorm]

```

learn does *n* steps of EM:

```

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

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

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

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

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

```

learnEM.py — (continued)
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
77     for c in range(self.num_classes):
78         res += prod(fc[i][feat(tple)][c]
79                    for (i,feat) in
80                      enumerate(feats))/(cc[c]**(len(feats)-1))
79     if res>0:
80         return -math.log2(res/len(self.dataset.train))
81     else:
82         return float("inf") #infinity

```

Figure 10.4 shows the training and test error for various numbers of classes for the carbool dataset (calls commented out at the end of the code).

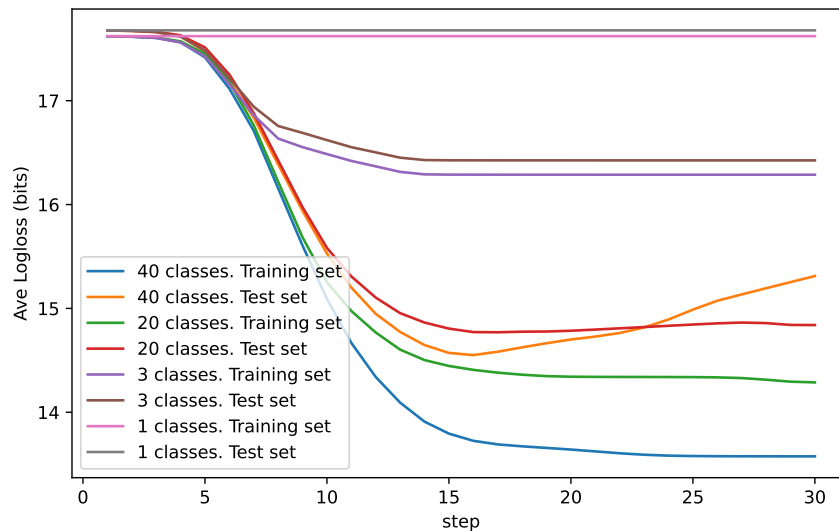


Figure 10.4: EM plotting error.

learnEM.py — (continued)

```

84 def plot_error(self, maxstep=20):
85     """Plots the logloss error as a function of the number of steps"""
86     plt.ion()
87     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):

```

```

108     """returns the product of the elements of L"""
109     res = 1
110     for e in L:
111         res *= e
112     return res
113
114 def random_dist(k):
115     """generate k random numbers that sum to 1"""
116     res = [random.random() for i in range(k)]
117     s = sum(res)
118     return [v/s for v in res]
119
120 data = Data_from_file('data/emdata2.csv', num_train=10, target_index=2000)
121 eml = EM_learner(data,2)
122 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', target_index=2000,
132     one_hot=True)
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(30) # 3 classes
137 # em3=EM_learner(data,20); em3.plot_error(30) # 20 classes
138 # em3=EM_learner(data,40); em3.plot_error(30) # 40 classes
139 # em3=EM_learner(data,1); em3.plot_error(30) # 1 classes (predict mean)

```

Exercise 10.2 For data where there are naturally 2 classes, does EM with 3 classes do better on the training set after a while than 2 classes? Is it better on a test set. Explain why. Hint: look what the 3 classes are. Use "eml.show_class(i)" for each of the classes $i \in [0, 3)$.

Exercise 10.3 Write code to plot the logloss as a function of the number of classes (from 1 to, say, 30) for a fixed number of iterations. (From the experience with the existing code, think about how many iterations are appropriate.

Exercise 10.4 Repeat the previous exercise, but use cross validation to select the number of iterations as a function of the number of classes and other features of the dataset.

Causality

11.1 Do Questions

A causal model can answer “do” questions.

The `intervene` function takes a belief network and a *variable : value* dictionary specifying what to “do”, and returns a belief network resulting from intervening to set each variable in the dictionary to its value specified. It replaces the conditional probability distribution, CPD, (Section 9.3) of each intervened variable with an constant CPD.

```
_____probDo.py — Probabilistic inference with the do operator_____
11 from probGraphicalModels import InferenceMethod, BeliefNetwork
12 from probFactors import CPD, ConstantCPD
13
14 def intervene(bn, do={}):
15     assert isinstance(bn, BeliefNetwork), f"Do only applies to belief
16         networks ({bn.title})"
17     if do=={}:
18         return bn
19     else:
20         newfacs = ({f for (ch,f) in bn.var2cpt.items() if ch not in do} |
21                     {ConstantCPD(v,c) for (v,c) in do.items()})
22         return BeliefNetwork(f"{bn.title}(do={do})", bn.variables, newfacs)
```

The following adds the `queryDo` method to the `InferenceMethod` class, so it can be used with any inference method. It replaces the graphical model with the modified one, runs the inference algorithm, and restores the initial belief network.

```
_____probDo.py — (continued) _____
23 def queryDo(self, qvar, obs={}, do={}):
```

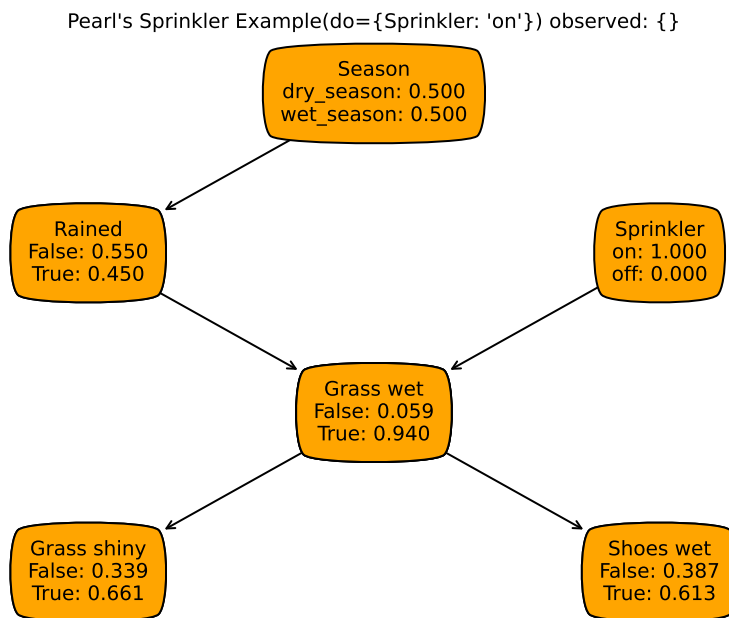


Figure 11.1: The sprinkler belief network with $\text{do}=\{\text{Sprinkler: "on"}\}$.

```

24     """Extends query method to also allow for interventions.
25     """
26     oldBN, self.gm = self.gm, intervene(self.gm, do)
27     result = self.query(qvar, obs)
28     self.gm = oldBN # restore original
29     return result
30
31 # make queryDo available for all inference methods
32 InferenceMethod.queryDo = queryDo

```

The following example is based on the sprinkler belief network of Section 9.4.2 shown in Figure 9.4. The network with the intervention of putting the sprinkler on is shown in Figure 11.1.

```

34 from probRC import ProbRC
35
36 from probExamples import bn_sprinkler, Season, Sprinkler, Rained,
    Grass_wet, Grass_shiny, Shoes_wet
37 bn_sprinklerv = ProbRC(bn_sprinkler)
38 ## bn_sprinklerv.queryDo(Shoes_wet)
39 ## bn_sprinklerv.queryDo(Shoes_wet, obs={Sprinkler: "on"})
40 ## bn_sprinklerv.queryDo(Shoes_wet, do={Sprinkler: "on"})
41 ## bn_sprinklerv.queryDo(Season, obs={Sprinkler: "on"})
42 ## bn_sprinklerv.queryDo(Season, do={Sprinkler: "on"})

```

Gateway Drug? observed: {}

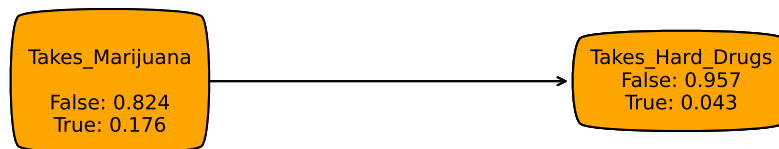


Figure 11.2: Does taking marijuana lead to hard drugs: observable variables

```

43
44 ### Showing posterior distributions:
45 # bn_sprinklerv.show_post({})
46 # bn_sprinklerv.show_post({Sprinkler:"on"})
47 # spon = intervene(bn_sprinkler, do={Sprinkler:"on"})
48 # ProbRC(spon).show_post({})
  
```

The following is a representation of a possible model where marijuana is a gateway drug to harder drugs (or not). Before reading the code, try the commented-out queries at the end. Figure 11.2 shows the network with the observable variables, Takes_Marijuana and Takes_Hard_Drugs.

```

_____probDo.py — (continued) _____
50 from variable import Variable
51 from probFactors import Prob
52 from probGraphicalModels import BeliefNetwork
53 boolean = [False, True]
54
55 Drug_Prone = Variable("Drug_Prone", boolean, position=(0.1,0.5)) #
    (0.5,0.9))
56 Side_Effects = Variable("Side_Effects", boolean, position=(0.1,0.5)) #
    (0.5,0.1))
57 Takes_Marijuana = Variable("\nTakes_Marijuana\n", boolean,
    position=(0.1,0.5))
58 Takes_Hard_Drugs = Variable("Takes_Hard_Drugs", boolean,
    position=(0.9,0.5))
59
60 p_dp = Prob(Drug_Prone, [], [0.8, 0.2])
61 p_be = Prob(Side_Effects, [Takes_Marijuana], [[1, 0], [0.4, 0.6]])
62 p_tm = Prob(Takes_Marijuana, [Drug_Prone], [[0.98, 0.02], [0.2, 0.8]])
63 p_thd = Prob(Takes_Hard_Drugs, [Side_Effects, Drug_Prone],
64             # Drug_Prone=False Drug_Prone=True
65             [[0.999, 0.001], [0.6, 0.4]], # Side_Effects=False
66             [[0.99999, 0.00001], [0.995, 0.005]]]) # Side_Effects=True
67
  
```

```

68 | drugs = BeliefNetwork("Gateway Drug?",
69 |                       [Drug_Prone, Side_Effects, Takes_Marijuana,
70 |                        Takes_Hard_Drugs],
71 |                       [p_tm, p_dp, p_be, p_thd])
72 |
73 | drugsq = ProbRC(drugs)
74 | # drugsq.queryDo(Takes_Hard_Drugs)
75 | # drugsq.queryDo(Takes_Hard_Drugs, obs = {Takes_Marijuana: True})
76 | # drugsq.queryDo(Takes_Hard_Drugs, obs = {Takes_Marijuana: False})
77 | # drugsq.queryDo(Takes_Hard_Drugs, do = {Takes_Marijuana: True})
78 | # drugsq.queryDo(Takes_Hard_Drugs, do = {Takes_Marijuana: False})
79 |
80 | # ProbRC(drugs).show_post({})
81 | # ProbRC(drugs).show_post({Takes_Marijuana: True})
82 | # ProbRC(drugs).show_post({Takes_Marijuana: False})
83 | # ProbRC(intervene(drugs, do={Takes_Marijuana: True})).show_post({})
84 | # ProbRC(intervene(drugs, do={Takes_Marijuana: False})).show_post({})
85 | # Why was that? Try the following then repeat:
86 | # Drug_Prone.position=(0.5,0.9); Side_Effects.position=(0.5,0.1)

```

11.2 Counterfactual Reasoning

The following provides two examples of counterfactual reasoning. In the following code, the user has to provide the deterministic system with noise. As we will see, there are multiple deterministic systems with noise that can produce the same causal probabilities.

```

_____probCounterfactual.py — Counterfactual Query Example_____
11 | from variable import Variable
12 | from probFactors import Prob, ProbDT, IFeq, SameAs, Dist
13 | from probGraphicalModels import BeliefNetwork
14 | from probRC import ProbRC
15 | from probDo import queryDo
16 |
17 | boolean = [False, True]

```

11.2.1 Choosing Deterministic System

This section presents an example to encourage you to think about what deterministic system to use.

Consider the following example (thanks to Sophie Song). Suppose Bob went on a date with Alice. Bob was either on time or not (variable *B* is true when Bob is on time). Alice, who is fastidious about punctuality chooses whether to go on a second date (variable *A* is true when Alice agrees to a second date). Whether Bob is late depends on which cab company he called (variable *C*). Suppose Bob calls one of the cab companies, he was late, and Alice doesn't ask for a second date. Bob wonders "what if I had called the other

CBA Counterfactual Example

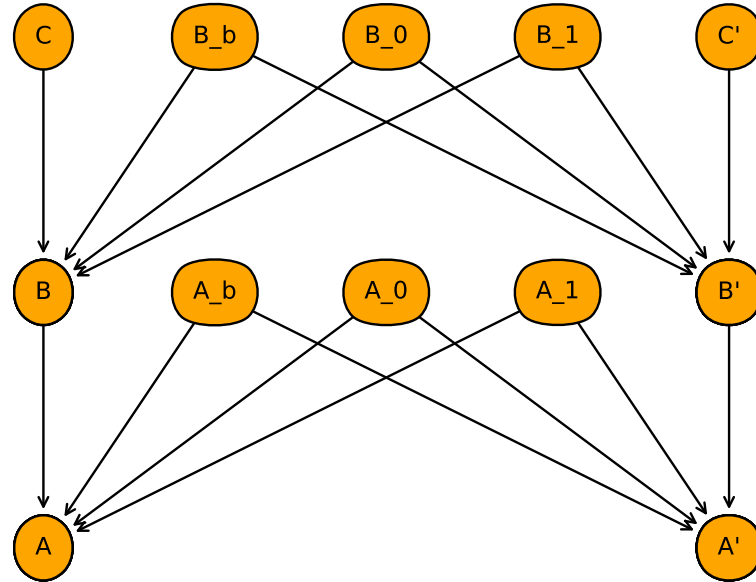


Figure 11.3: $C \rightarrow B \rightarrow A$ belief network for “what if C ”. Figure generated by `cbaCounter.show()`

cab company”. Suppose all variables are Boolean. C causally depends on B , and not directly on C , and B depends on C , so the appropriate causal model is $C \rightarrow B \rightarrow A$.

Assume the following probabilities obtained from observations (where the lower case c represents $C = \text{true}$, and similarly for other variables):

$$P(c) = 0.5$$

$$P(b \mid c) = P(b \mid \neg c) = 0.7 \quad (\text{the cab companies are equally reliable})$$

$$(a \mid b) = 0.4, (a \mid \neg b) = 0.2.$$

Consider “what if C was True” or “what if C was False”. For example, suppose $A = \text{false}$ and $C = \text{false}$ is observed and you want the probability of A if C were false.

Figure 11.3 shows the paired network for “what if C ”. The primed variables represent the situation where C is counterfactually True or False. In this network, C_{prime} should be conditioned on. Conditioning on C_{prime} should not affect the non-primed variables. (You should check this).

```

probCounterfactual.py — (continued)
19 # as a deterministic system with independent noise
20 C = Variable("C", boolean, position=(0.1,0.8))
21 B = Variable("B", boolean, position=(0.1,0.4))
22 A = Variable("A", boolean, position=(0.1,0.0))
23 Cprime = Variable("C'", boolean, position=(0.9,0.8))
24 Bprime = Variable("B'", boolean, position=(0.9,0.4))
25 Aprime = Variable("A'", boolean, position=(0.9,0.0))
26 B_b = Variable("B_b", boolean, position=(0.3,0.8))
27 B_0 = Variable("B_0", boolean, position=(0.5,0.8))
28 B_1 = Variable("B_1", boolean, position=(0.7,0.8))
29 A_b = Variable("A_b", boolean, position=(0.3,0.4))
30 A_0 = Variable("A_0", boolean, position=(0.5,0.4))
31 A_1 = Variable("A_1", boolean, position=(0.7,0.4))

```

The conditional probability $P(A \mid B)$ is represented using three noise parameters, A_b , A_0 and A_1 , with the equivalence:

$$a \equiv a_b \vee (\neg b \wedge a_0) \vee (b \wedge a_1)$$

Thus a_b is the background cause of a , a_0 is the cause used when $B=\text{false}$ and a_1 is the cause used when $B=\text{true}$. Note that this is over parametrized with respect the belief network, using three parameters whereas arbitrary conditional probability can be represented using two parameters.

The running example where $(a \mid b) = 0.4$ and $(a \mid \neg b) = 0.2$ can be represented using

$$P(a_b) = 0, P(a_0) = 0.2, P(a_1) = 0.4$$

or

$$P(a_b) = 0.2, P(a_0) = 0, P(a_1) = 0.25$$

(and infinitely many others between these). These cannot be distinguished by observations or by interventions. As you can see if you play with the code, these have different counterfactual conclusions.

$P(B \mid C)$ is represented similarly, using variables B_b , B_0 , and B_1 .

The following code uses the decision tree representation of conditional probabilities of Section 9.3.4.

```

probCounterfactual.py — (continued)
33 p_C = Prob(C, [], [0.5,0.5])
34 p_B = ProbDT(B, [C, B_b, B_0, B_1], IFeq(B_b,True,Dist([0,1]),
35                                     IFeq(C,True,SameAs(B_1),SameAs(B_0))))
36 p_A = ProbDT(A, [B, A_b, A_0, A_1], IFeq(A_b,True,Dist([0,1]),
37                                     IFeq(B,True,SameAs(A_1),SameAs(A_0))))
38 p_Cprime = Prob(Cprime,[], [0.5,0.5])
39 p_Bprime = ProbDT(Bprime, [Cprime, B_b, B_0, B_1],
40                       IFeq(B_b,True,Dist([0,1]),

```

```

41         IFeq(Cprime, True, SameAs(B_1), SameAs(B_0))))
42 p_Aprime = ProbDT(Aprime, [Bprime, A_b, A_0, A_1],
43         IFeq(A_b, True, Dist([0,1]),
44         IFeq(Bprime, True, SameAs(A_1), SameAs(A_0))))
45 p_b_b = Prob(B_b, [], [1,0])
46 p_b_0 = Prob(B_0, [], [0.3,0.7])
47 p_b_1 = Prob(B_1, [], [0.3,0.7])
48
49 p_a_b = Prob(A_b, [], [1,0])
50 p_a_0 = Prob(A_0, [], [0.8,0.2])
51 p_a_1 = Prob(A_1, [], [0.6,0.4])
52
53 p_b_np = Prob(B, [], [0.3,0.7]) # for AB network
54 p_Bprime_np = Prob(B, [], [0.3,0.7]) # for AB network
55 ab_Counter = BeliefNetwork("AB Counterfactual Example",
56         [A,B,Aprime,Bprime, A_b,A_0,A_1],
57         [p_A, p_b_np, p_Aprime, p_Bprime_np, p_a_b, p_a_0,
58         p_a_1])
59 cbaCounter = BeliefNetwork("CBA Counterfactual Example",
60         [A,B,C, Aprime,Bprime,Cprime, B_b,B_0,B_1, A_b,A_0,A_1],
61         [p_A, p_B, p_C, p_Aprime, p_Bprime, p_Cprime,
62         p_b_b, p_b_0, p_b_1, p_a_b, p_a_0, p_a_1])

```

Here are some queries you might like to try. The `show_post` queries might be most useful if you have the space to show multiple queries.

```

_____probCounterfactual.py — (continued) _____
64 cbaq = ProbRC(cbaCounter)
65 # cbaq.queryDo(Aprime, obs = {C:True, Cprime:False})
66 # cbaq.queryDo(Aprime, obs = {C:False, Cprime:True})
67 # cbaq.queryDo(Aprime, obs = {A:True, C:True, Cprime:False})
68 # cbaq.queryDo(Aprime, obs = {A:False, C:True, Cprime:False})
69 # cbaq.queryDo(Aprime, obs = {A:False, C:True, Cprime:False})
70 # cbaq.queryDo(A_1, obs = {C:True, Aprime:False})
71 # cbaq.queryDo(A_0, obs = {C:True, Aprime:False})
72
73 # cbaq.show_post(obs = {})
74 # cbaq.show_post(obs = {C:True, Cprime:False})
75 # cbaq.show_post(obs = {A:False, C:True, Cprime:False})
76 # cbaq.show_post(obs = {A:True, C:True, Cprime:False})

```

Exercise 11.1 Consider the scenario “Bob called the first cab ($C = \text{true}$), was late and Alice agrees to a second date”. What would you expect from the scenario “what if Bob called the other cab?”. What does the network predict? Design probabilities for the noise variables that fits the conditional probability and also fits your expectation.

Exercise 11.2 How would you expect the counterfactual conclusion to change given the following two scenarios that fit the story:

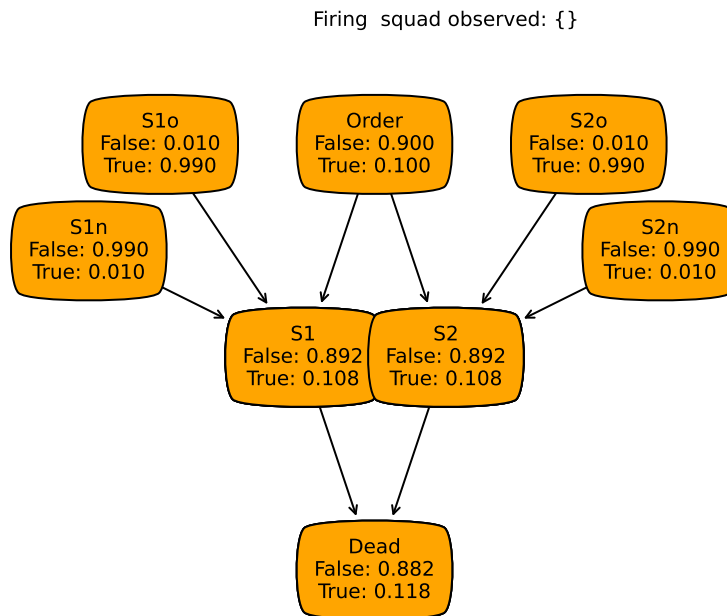


Figure 11.4: Firing squad belief network (figure obtained from `fsq.show_post({})`)

- The cabs are both very reliable and start at the same location (and so face the same traffic).
 - The cabs are each 90% reliable and start from opposite directions.
- (a) How would you expect the predictions to differ in these two cases?
 - (b) How can you fit the conditional probabilities above and represent each of these by changing the probabilities of the noise variables?
 - (c) How can these be learned from data? (Hint: consider learning a correlation between the taxi arrivals). Is your approach always applicable? If not, for which cases is it applicable or not.

11.2.2 Firing Squad Example

The following is the firing squad example of Pearl [2009] as a deterministic system. See Figure 11.4.

```

probCounterfactual.py — (continued)
78 Order = Variable("Order", boolean, position=(0.4,0.8))
79 S1 = Variable("S1", boolean, position=(0.3,0.4))
80 S1o = Variable("S1o", boolean, position=(0.1,0.8))
81 S1n = Variable("S1n", boolean, position=(0.0,0.6))
82 S2 = Variable("S2", boolean, position=(0.5,0.4))
83 S2o = Variable("S2o", boolean, position=(0.7,0.8))

```



```

84 S2n = Variable("S2n", boolean, position=(0.8,0.6))
85 Dead = Variable("Dead", boolean, position=(0.4,0.0))

```

Instead of the tabular representation of the if-then-else structure used for the $A \rightarrow B \rightarrow C$ network above, the following uses the decision tree representation of conditional probabilities of Section 9.3.4.

```

_____probCounterfactual.py — (continued)_____
87 p_S1 = ProbDT(S1, [Order, S1o, S1n],
88               IFeq(Order, True, SameAs(S1o), SameAs(S1n)))
89 p_S2 = ProbDT(S2, [Order, S2o, S2n],
90               IFeq(Order, True, SameAs(S2o), SameAs(S2n)))
91 p_dead = Prob(Dead, [S1, S2], [[[1,0],[0,1]],[[0,1],[0,1]]])
92         #IFeq(S1, True, True, SameAs(S2)))
93 p_order = Prob(Order, [], [0.9, 0.1])
94 p_s1o = Prob(S1o, [], [0.01, 0.99])
95 p_s1n = Prob(S1n, [], [0.99, 0.01])
96 p_s2o = Prob(S2o, [], [0.01, 0.99])
97 p_s2n = Prob(S2n, [], [0.99, 0.01])
98
99 firing_squad = BeliefNetwork("Firing squad",
100                             [Order, S1, S1o, S1n, S2, S2o, S2n, Dead],
101                             [p_order, p_dead, p_S1, p_s1o, p_s1n, p_S2, p_s2o,
                               p_s2n])
102 fsq = ProbRC(firing_squad)
103 # fsq.queryDo(Dead)
104 # fsq.queryDo(Order, obs={Dead:True})
105 # fsq.queryDo(Dead, obs={Order:True})
106 # fsq.show_post({})
107 # fsq.show_post({Dead:True})
108 # fsq.show_post({S2:True})

```

Exercise 11.3 Create the network for “what if shooter 2 did or did not shoot”. Give the probabilities of the following counterfactuals:

- The prisoner is dead; what is the probability that the prisoner would be dead if shooter 2 did not shoot?
- Shooter 2 shot; what is the probability that the prisoner would be dead if shooter 2 did not shoot?
- No order was given, but the prisoner is dead; what is the probability that the prisoner would be dead if shooter 2 did not shoot?

Exercise 11.4 Create the network for “what if the order was or was not given”. Give the probabilities of the following counterfactuals:

- The prisoner is dead; what is the probability that the prisoner would be dead if the order was not given?
- The prisoner is not dead; what is the probability that the prisoner would be dead if the order was not given? (Is this different from the prior that the prisoner is dead, or the posterior that the prisoner was dead given the order was not given).

- (c) Shooter 2 shot; what is the probability that the prisoner would be dead if the order was not given?
- (d) Shooter 2 did not shoot; what is the probability that the prisoner would be dead if the order was given? (Is this different from the probability that the prisoner would be dead if the order was given without the counterfactual observation)?

Planning with Uncertainty

12.1 Decision Networks

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

First, define factors that define the utility. Here the **utility** is a function of the variables in *vars*. In a **utility table** the utility is defined in terms of a tabular factor – a list that enumerates the values – as in Section 9.3.3. Another representations for factors (Section 9.2) could able be used.

```
-----decnNetworks.py — Representations for Decision Networks-----
11 from probGraphicalModels import GraphicalModel, BeliefNetwork
12 from probFactors import Factor, CPD, TabFactor, factor_times, Prob
13 from variable 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, name="Utility")
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 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. Note that this is an instance of BeliefNetwork but overrides `__init__`.

```

----- decnNetworks.py --- (continued) -----
35 class DecisionNetwork(BeliefNetwork):
36     def __init__(self, title, vars, factors):
37         """title is a string
38         vars is a list of variables (random and decision)
39         factors is a list of factors (instances of CPD and Utility)
40         """
41         GraphicalModel.__init__(self, title, vars, factors)
42         # not BeliefNetwork.__init__
43         self.var2parents = ({v : v.parents for v in vars
44                             if isinstance(v,DecisionVariable)})
45                             | {f.child:f.parents for f in factors
46                             if isinstance(f,CPD)})
47         self.children = {n:[] for n in self.variables}
48         for v in self.var2parents:
49             for par in self.var2parents[v]:
50                 self.children[par].append(v)
51         self.utility_factor = [f for f in factors
52                                if isinstance(f,Utility)][0]
53         self.topological_sort_saved = None
54
55     def __str__(self):
56         return self.title

```

The split order ensures that the parents of a decision node are split before the decision node, and no other variables (if that is possible).

```

----- decnNetworks.py --- (continued) -----
58 def split_order(self):
59     so = []
60     tops = self.topological_sort()
61     for v in tops:
62         if isinstance(v,DecisionVariable):
63             so += [p for p in v.parents if p not in so]
64             so.append(v)
65     so += [v for v in tops if v not in so]
66     return so

```

```

decnNetworks.py — (continued)
68 def show(self, fontsize=10,
69         colors={'utility':'red', 'decision':'lime', 'random':'orange'}):
70     plt.ion() # interactive
71     ax = plt.figure().gca()
72     ax.set_axis_off()
73     plt.title(self.title, fontsize=fontsize)
74     for par in self.utility_factor.variables:
75         ax.annotate("Utility", par.position,
76                    xytext=self.utility_factor.position,
77                    arrowprops={'arrowstyle':'<-'},
78                    bbox=dict(boxstyle="sawtooth,pad=1.0",
79                            facecolor=colors['utility']),
80                    ha='center', va='center', fontsize=fontsize)
81     for var in reversed(self.topological_sort()):
82         if isinstance(var, DecisionVariable):
83             bbox = dict(boxstyle="square,pad=1.0",
84                         facecolor=colors['decision'])
85         else:
86             bbox = dict(boxstyle="round4,pad=1.0,rounding_size=0.5",
87                         facecolor=colors['random'])
88         if self.var2parents[var]:
89             for par in self.var2parents[var]:
90                 ax.annotate(var.name, par.position, xytext=var.position,
91                            arrowprops={'arrowstyle':'<-'},bbox=bbox,
92                            ha='center', va='center',
93                            fontsize=fontsize)
94         else:
95             x,y = var.position
96             plt.text(x,y,var.name,bbox=bbox,ha='center', va='center',
97                     fontsize=fontsize)

```

12.1.1 Example Decision Networks

Umbrella Decision Network

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

```

decnNetworks.py — (continued)
98 Weather = Variable("Weather", ["NoRain", "Rain"],
99                      position=(0.5,0.8))
100 Forecast = Variable("Forecast", ["Sunny", "Cloudy", "Rainy"],
101                      position=(0,0.4))
102 # Each variant uses one of the following:
103 Umbrella = DecisionVariable("Umbrella", ["Take", "Leave"], {Forecast},
104                             position=(0.5,0))
105
106 p_weather = Prob(Weather, [], {"NoRain":0.7, "Rain":0.3})
107 p_forecast = Prob(Forecast, [Weather],

```

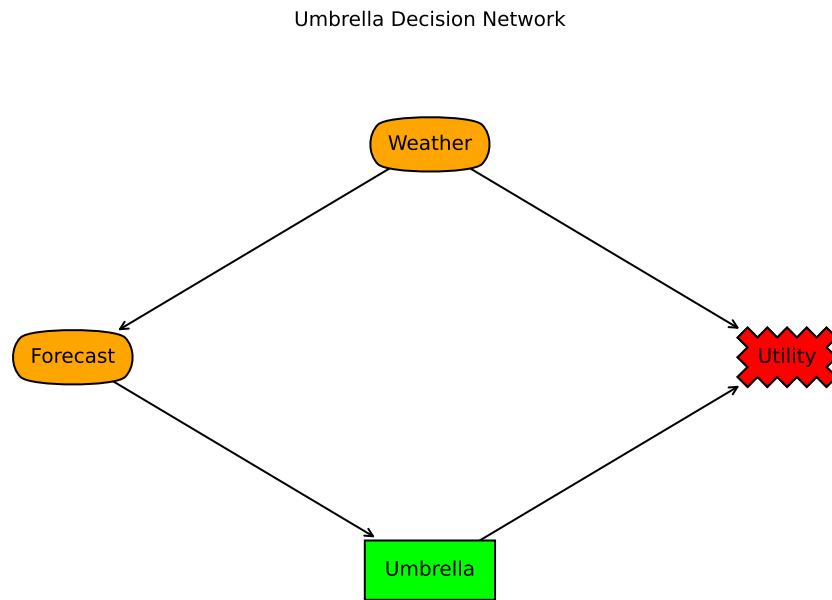


Figure 12.1: The umbrella decision network. Figure generated by `umbrella_dn.show()`

```

108         {"NoRain":{"Sunny":0.7, "Cloudy":0.2, "Rainy":0.1},
109          "Rain":{"Sunny":0.15, "Cloudy":0.25, "Rainy":0.6}})
110 umb_utility = UtilityTable([Weather, Umbrella],
111                             {"NoRain":{"Take":20, "Leave":100},
112                              "Rain":{"Take":70, "Leave":0}}, position=(1,0.4))
113
114 umbrella_dn = DecisionNetwork("Umbrella Decision Network",
115                               [Weather, Forecast, Umbrella],
116                               {p_weather, p_forecast, umb_utility})
117
118 # umbrella_dn.show()
119 # umbrella_dn.show(fontsize=15)

```

The following is a variant with the umbrella decision having 2 parents; nothing else has changed. This is interesting because one of the parents is not needed; if the agent knows the weather, it can ignore the forecast.

```

121 Umbrella2p = DecisionVariable("Umbrella", ["Take", "Leave"],
122                                {Forecast, Weather}, position=(0.5,0))
123 umb_utility2p = UtilityTable([Weather, Umbrella2p],
124                               {"NoRain":{"Take":20, "Leave":100},
125                                "Rain":{"Take":70, "Leave":0}},
126                               position=(1,0.4))

```

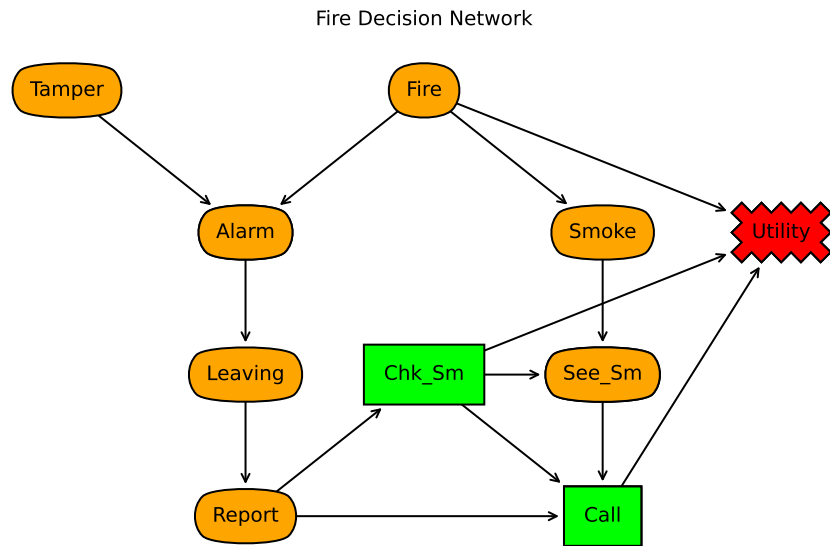


Figure 12.2: Fire Decision Network. Figure generated by `fire_dn.show()`

```

127 | umbrella_dn2p = DecisionNetwork("Umbrella Decision Network (extra arc)",
128 |                               {Weather, Forecast, Umbrella2p},
129 |                               {p_weather, p_forecast, umb_utility2p})
130 |
131 | # umbrella_dn2p.show()
132 | # umbrella_dn2p.show(fontsize=15)

```

Fire Decision Network

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

```

----- decnNetworks.py --- (continued) -----
134 | boolean = [False, True]
135 | Alarm = Variable("Alarm", boolean, position=(0.25,0.633))
136 | Fire = Variable("Fire", boolean, position=(0.5,0.9))
137 | Leaving = Variable("Leaving", boolean, position=(0.25,0.366))
138 | Report = Variable("Report", boolean, position=(0.25,0.1))
139 | Smoke = Variable("Smoke", boolean, position=(0.75,0.633))
140 | Tamper = Variable("Tamper", boolean, position=(0,0.9))
141 |
142 | See_Sm = Variable("See_Sm", boolean, position=(0.75,0.366) )
143 | Chk_Sm = DecisionVariable("Chk_Sm", boolean, {Report},

```

```

144         position=(0.5, 0.366))
145 Call = DecisionVariable("Call", boolean,{See_Sm,Chk_Sm,Report},
146         position=(0.75,0.1))
147
148 f_ta = Prob(Tamper,[],[0.98,0.02])
149 f-fi = Prob(Fire,[],[0.99,0.01])
150 f-sm = Prob(Smoke,[Fire],[[0.99,0.01],[0.1,0.9]])
151 f-al = Prob(Alarm,[Fire,Tamper],[[0.9999, 0.0001], [0.15, 0.85]],
152         [[0.01, 0.99], [0.5, 0.5]])
153 f-lv = Prob(Leaving,[Alarm],[[0.999, 0.001], [0.12, 0.88]])
154 f-re = Prob(Report,[Leaving],[[0.99, 0.01], [0.25, 0.75]])
155 f-ss = Prob(See_Sm,[Chk_Sm,Smoke],[[1,0],[1,0]],[[1,0],[0,1]])
156
157 ut = UtilityTable([Chk_Sm,Fire,Call],
158         [[0,-200],[-5000,-200]],[[-20,-220],[-5020,-220]],
159         position=(1,0.633))
160
161 fire_dn = DecisionNetwork("Fire Decision Network",
162         {Tamper,Fire,Alarm,Leaving,Smoke,Call,See_Sm,Chk_Sm,Report},
163         {f_ta,f-fi,f-sm,f-al,f-lv,f-re,f-ss,ut})
164
165 # print(ut.to_table())
166 # fire_dn.show()
167 # fire_dn.show(fontsize=15)

```

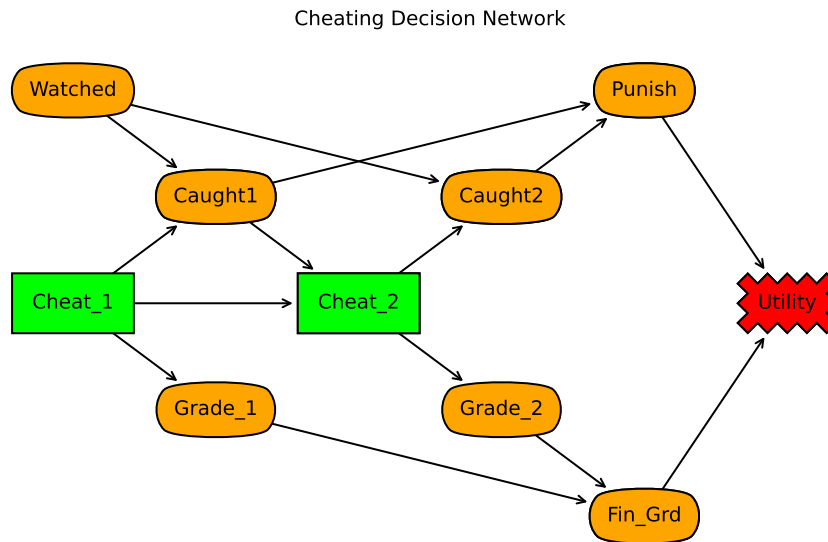
Cheating Decision Network

The following is the representation of the cheating decision shown in Figure 12.3. Someone has to decide whether to cheat at two different times. Cheating can improve grades. However, someone is watching for cheating, and if caught, results in punishment. The utility is a combination of final grade and the punishment. The decision maker finds out whether they were caught the first time when they have to decide whether to cheat the second time.

```

decnNetworks.py — (continued)
169 grades = ['A','B','C','F']
170 Watched = Variable("Watched", boolean, position=(0,0.9))
171 Caught1 = Variable("Caught1", boolean, position=(0.2,0.7))
172 Caught2 = Variable("Caught2", boolean, position=(0.6,0.7))
173 Punish = Variable("Punish", ["None","Suspension","Recorded"],
174         position=(0.8,0.9))
175 Grade_1 = Variable("Grade_1", grades, position=(0.2,0.3))
176 Grade_2 = Variable("Grade_2", grades, position=(0.6,0.3))
177 Fin_Grd = Variable("Fin_Grd", grades, position=(0.8,0.1))
178 Cheat_1 = DecisionVariable("Cheat_1", boolean, set(), position=(0,0.5))
179 Cheat_2 = DecisionVariable("Cheat_2", boolean, {Cheat_1,Caught1},
180         position=(0.4,0.5))
181
182 p-wa = Prob(Watched,[],[0.7, 0.3])

```


Figure 12.3: Cheating Decision Network (`cheating-dn.show()`)

```

183 p_cc1 = Prob(Caught1,[Watched,Cheat_1],[[1.0, 0.0], [0.9, 0.1]],
184             [[1.0, 0.0], [0.5, 0.5]])
185 p_cc2 = Prob(Caught2,[Watched,Cheat_2],[[1.0, 0.0], [0.9, 0.1]],
186             [[1.0, 0.0], [0.5, 0.5]])
187 p_pun = Prob(Punish,[Caught1,Caught2],
188             [{"None":0,"Suspension":0,"Recorded":0},
189              {"None":0.5,"Suspension":0.4,"Recorded":0.1}],
190             [{"None":0.6,"Suspension":0.2,"Recorded":0.2},
191              {"None":0.2,"Suspension":0.3,"Recorded":0.3}])
192 p_gr1 = Prob(Grade_1,[Cheat_1], [{"A":0.2, 'B':0.3, 'C':0.3, 'F': 0.2},
193                                  {'A':0.5, 'B':0.3, 'C':0.2, 'F':0.0}])
194 p_gr2 = Prob(Grade_2,[Cheat_2], [{"A":0.2, 'B':0.3, 'C':0.3, 'F': 0.2},
195                                  {'A':0.5, 'B':0.3, 'C':0.2, 'F':0.0}])
196 p_fg = Prob(Fin_Grd,[Grade_1,Grade_2],
197             {'A':{'A':{'A':1.0, 'B':0.0, 'C': 0.0, 'F':0.0},
198                  'B': {'A':0.5, 'B':0.5, 'C': 0.0, 'F':0.0},
199                  'C':{'A':0.25, 'B':0.5, 'C': 0.25, 'F':0.0},
200                  'F':{'A':0.25, 'B':0.25, 'C': 0.25, 'F':0.25}},
201              'B':{'A':{'A':0.5, 'B':0.5, 'C': 0.0, 'F':0.0},
202                  'B': {'A':0.0, 'B':1, 'C': 0.0, 'F':0.0},
203                  'C':{'A':0.0, 'B':0.5, 'C': 0.5, 'F':0.0},
204                  'F':{'A':0.0, 'B':0.25, 'C': 0.5, 'F':0.25}},
205              'C':{'A':{'A':0.25, 'B':0.5, 'C': 0.25, 'F':0.0},
206                  'B': {'A':0.0, 'B':0.5, 'C': 0.5, 'F':0.0},

```

```

207         'C': {'A': 0.0, 'B': 0.0, 'C': 1, 'F': 0.0},
208         'F': {'A': 0.0, 'B': 0.0, 'C': 0.5, 'F': 0.5}},
209     'F': {'A': {'A': 0.25, 'B': 0.25, 'C': 0.25, 'F': 0.25},
210           'B': {'A': 0.0, 'B': 0.25, 'C': 0.5, 'F': 0.25},
211           'C': {'A': 0.0, 'B': 0.0, 'C': 0.5, 'F': 0.5},
212           'F': {'A': 0.0, 'B': 0.0, 'C': 0, 'F': 1.0}}})
213
214     utc = UtilityTable([Punish, Fin_Grd],
215                        {'None': {'A': 100, 'B': 90, 'C': 70, 'F': 50},
216                          'Suspension': {'A': 40, 'B': 20, 'C': 10, 'F': 0},
217                          'Recorded': {'A': 70, 'B': 60, 'C': 40, 'F': 20}},
218                        position=(1, 0.5))
219
220     cheating_dn = DecisionNetwork("Cheating Decision Network",
221                                  {Punish, Caught2, Watched, Fin_Grd, Grade_2, Grade_1, Cheat_2, Caught1, Cheat_1},
222                                  {p_wa, p_cc1, p_cc2, p_pun, p_gr1, p_gr2, p_fg, utc})
223
224     # cheating_dn.show()
225     # cheating_dn.show(fontsize=15)

```

Chain of 3 decisions

The following decision network represents a finite-stage fully-observable Markov decision process with a single reward (utility) at the end. It is interesting because the parents do not include all the predecessors. The methods we use will work without change on this, even though the agent does not condition on all of its previous observations and actions. The output of `ch3.show()` is shown in Figure 12.4.

```

----- decnNetworks.py --- (continued) -----
227 S0 = Variable('S0', boolean, position=(0, 0.5))
228 D0 = DecisionVariable('D0', boolean, {S0}, position=(1/7, 0.1))
229 S1 = Variable('S1', boolean, position=(2/7, 0.5))
230 D1 = DecisionVariable('D1', boolean, {S1}, position=(3/7, 0.1))
231 S2 = Variable('S2', boolean, position=(4/7, 0.5))
232 D2 = DecisionVariable('D2', boolean, {S2}, position=(5/7, 0.1))
233 S3 = Variable('S3', boolean, position=(6/7, 0.5))
234
235 p_s0 = Prob(S0, [], [0.5, 0.5])
236 tr = [[[0.1, 0.9], [0.9, 0.1]], [[0.2, 0.8], [0.8, 0.2]]] # 0 is flip, 1
      is keep value
237 p_s1 = Prob(S1, [D0, S0], tr)
238 p_s2 = Prob(S2, [D1, S1], tr)
239 p_s3 = Prob(S3, [D2, S2], tr)
240
241 ch3U = UtilityTable([S3], [0, 1], position=(7/7, 0.9))
242
243 ch3 = DecisionNetwork("3-chain",
                       {S0, D0, S1, D1, S2, D2, S3}, {p_s0, p_s1, p_s2, p_s3, ch3U})

```

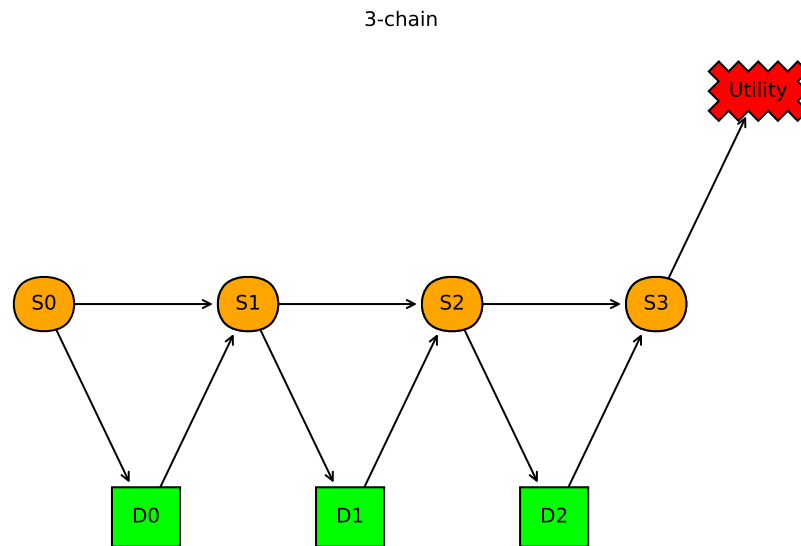


Figure 12.4: A decision network that is a chain of 3 decisions (`ch3.show()`)

```

244
245 # ch3.show()
246 # ch3.show(fontsize=15)

```

12.1.2 Decision Functions

The output of an optimization function is an optimal policy and its expected value. A policy is a list of decision functions. A decision function is the action for each decision variable as a function of its parents.

Let's represent the factor for a decision function as a dictionary.

```

_____decnNetworks.py — (continued)_____
248 class DictFactor(Factor):
249     """A factor that represents its values using a dictionary"""
250     def __init__(self, *pargs, **kwargs):
251         self.values = {}
252         Factor.__init__(self, *pargs, **kwargs)
253
254     def assign(self, assignment, value):
255         self.values[frozenset(assignment.items())] = value
256
257     def get_value(self, assignment):
258         ass = frozenset(assignment.items())

```

```

259         assert ass in self.values, f"assignment {assignment} cannot be
           evaluated"
260         return self.values[ass]
261
262 class DecisionFunction(DictFactor):
263     def __init__(self, decision, parents):
264         """ A decision function
265         decision is a decision variable
266         parents is a set of variables
267         """
268         self.decision = decision
269         self.parent = parents
270         DictFactor.__init__(self, parents, name=decision.name)

```

12.1.3 Recursive Conditioning for Decision Networks

An instance of a RC_DN object takes in a decision network. The query method uses recursive conditioning to compute the expected utility of the optimal policy. When it is finished, `self.opt_policy` is the optimal policy.

```

decnNetworks.py — (continued)
272 import math
273 from display import Displayable
274 from probGraphicalModels import GraphicalModel
275 from probFactors import Factor
276 from probRC import connected_components
277
278 class RC_DN(Displayable):
279     """The class that finds the optimal policy for a decision network.
280
281     dn is graphical model to query
282     """
283
284     def __init__(self, dn):
285         self.dn = dn
286         self.cache = {(frozenset(), frozenset()):1}
287         ## self.max_display_level = 3
288
289     def optimize(self, split_order=None, algorithm=None):
290         """computes expected utility, and creates optimal decision
           functions, where
291         elim_order is a list of the non-observed non-query variables in dn
292         algorithm is the (search algorithm to use). Default is self.rc
293         """
294         if algorithm is None:
295             algorithm = self.rc
296         if split_order == None:
297             split_order = self.dn.split_order()
298         self.opt_policy = {v:DecisionFunction(v, v.parents)
299                           for v in self.dn.variables}

```

```

300         if isinstance(v, DecisionVariable)}
301     return algorithm({}, self.dn.factors, split_order)
302
303     def show_policy(self):
304         print('\n'.join(df.to_table() for df in self.opt_policy.values()))

```

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

```

decnNetworks.py — (continued)
306     def rc0(self, context, factors, split_order):
307         """simplest search algorithm
308         context is a variable:value dictionary
309         factors is a set of factors
310         split_order is a list of variables in factors that are not in
311             context
312         """
313         self.display(3, "calling rc0,", (context, factors), "with
314             S0", split_order)
315         if not factors:
316             return 1
317         elif to_eval := {fac for fac in factors if
318             fac.can_evaluate(context)}:
319             self.display(3, "rc0 evaluating factors", to_eval)
320             val = math.prod(fac.get_value(context) for fac in to_eval)
321             return val * self.rc0(context, factors-to_eval, split_order)
322         else:
323             var = split_order[0]
324             self.display(3, "rc0 branching on", var)
325             if isinstance(var, DecisionVariable):
326                 assert set(context) <= set(var.parents), f"cannot optimize
327                     {var} in context {context}"
328                 maxres = -math.inf
329                 for val in var.domain:
330                     self.display(3, "In rc0, branching on", var, "=", val)
331                     newres = self.rc0({var:val}|context, factors,
332                         split_order[1:])
333                     if newres > maxres:
334                         maxres = newres
335                         theval = val
336                     self.opt_policy[var].assign(context, theval)
337                 return maxres
338             else:
339                 total = 0
340                 for val in var.domain:
341                     total += self.rc0({var:val}|context, factors,
342                         split_order[1:])
343                 self.display(3, "rc0 branching on", var, "returning", total)

```

```
338 |         return total
```

We can combine the optimization for decision networks above, with the improvements of recursive conditioning used for graphical models (Section 9.7, page 222).

```

decnNetworks.py — (continued)
340 | def rc(self, context, factors, split_order):
341 |     """ returns the number sum_{split_order} prod_{factors} given
           assignments in context
342 |     context is a variable:value dictionary
343 |     factors is a set of factors
344 |     split_order is a list of variables in factors that are not in
           context
345 |     """
346 |     self.display(3,"calling rc,", (context,factors))
347 |     ce = (frozenset(context.items()), frozenset(factors)) # key for the
           cache entry
348 |     if ce in self.cache:
349 |         self.display(2,"rc cache lookup", (context,factors))
350 |         return self.cache[ce]
351 | #     if not factors: # no factors; needed if you don't have forgetting
           and caching
352 | #         return 1
353 |     elif vars_not_in_factors := {var for var in context
           if not any(var in fac.variables for
354 |                     fac in factors)}:
355 |         # forget variables not in any factor
356 |         self.display(3,"rc forgetting variables", vars_not_in_factors)
357 |         return self.rc({key:val for (key,val) in context.items()
           if key not in vars_not_in_factors},
358 |                        factors, split_order)
359 |     elif to_eval := {fac for fac in factors if
           fac.can_evaluate(context)}:
360 |         # evaluate factors when all variables are assigned
361 |         self.display(3,"rc evaluating factors",to_eval)
362 |         val = math.prod(fac.get_value(context) for fac in to_eval)
363 |         if val == 0:
364 |             return 0
365 |         else:
366 |             return val * self.rc(context, {fac for fac in factors if fac
           not in to_eval}, split_order)
367 |     elif len(comp := connected_components(context, factors,
           split_order)) > 1:
368 |         # there are disconnected components
369 |         self.display(2,"splitting into connected components",comp)
370 |         return(math.prod(self.rc(context,f,eo) for (f,eo) in comp))
371 |     else:
372 |         assert split_order, f"split_order empty rc({context},{factors})"
373 |         var = split_order[0]
374 |         self.display(3, "rc branching on", var)
375 |

```

```

376         if isinstance(var, DecisionVariable):
377             assert set(context) <= set(var.parents), f"cannot optimize
               {var} in context {context}"
378             maxres = -math.inf
379             for val in var.domain:
380                 self.display(3, "In rc, branching on", var, "=", val)
381                 newres = self.rc({var:val}|context, factors,
               split_order[1:])
382                 if newres > maxres:
383                     maxres = newres
384                     theval = val
385                 self.opt_policy[var].assign(context, theval)
386                 self.cache[ce] = maxres
387             return maxres
388         else:
389             total = 0
390             for val in var.domain:
391                 total += self.rc({var:val}|context, factors,
               split_order[1:])
392             self.display(3, "rc branching on", var, "returning", total)
393             self.cache[ce] = total
394             return total

```

Here is how to run the optimizer on the example decision networks:

```

decnNetworks.py — (continued)
396 # Umbrella decision network
397 #urc = RC_DN(umbrella_dn)
398 #urc.optimize(algorithm=urc.rc0) #RC0
399 #urc.optimize() #RC
400 #urc.show_policy()
401
402 #rc_fire = RC_DN(fire_dn)
403 #rc_fire.optimize()
404 #rc_fire.show_policy()
405
406 #rc_cheat = RC_DN(cheating_dn)
407 #rc_cheat.optimize()
408 #rc_cheat.show_policy()
409
410 #rc_ch3 = RC_DN(ch3)
411 #rc_ch3.optimize()
412 #rc_ch3.show_policy()
413 # rc_ch3.optimize(algorithm=rc_ch3.rc0) # why does that happen?

```

12.1.4 Variable elimination for decision networks

VE_DN is variable elimination for decision networks. The method *optimize* is used to optimize all the decisions. Note that *optimize* requires a legal elimination ordering of the random and decision variables, otherwise it will give an

exception. (A decision node can only be maximized if the variables that are not its parents have already been eliminated.)

```

415 from probVE import VE
416
417 class VE_DN(VE):
418     """Variable Elimination for Decision Networks"""
419     def __init__(self, dn=None):
420         """dn is a decision network"""
421         VE.__init__(self, dn)
422         self.dn = dn
423
424     def optimize(self, elim_order=None, obs={}):
425         if elim_order == None:
426             elim_order = reversed(self.dn.split_order())
427         self.opt_policy = {}
428         proj_factors = [self.project_observations(fac, obs)
429                         for fac in self.dn.factors]
430         for v in elim_order:
431             if isinstance(v, DecisionVariable):
432                 to_max = [fac for fac in proj_factors
433                           if v in fac.variables and set(fac.variables) <=
434                               v.all_vars]
435                 assert len(to_max) == 1, "illegal variable order
436                     "+str(elim_order)+" at "+str(v)
437                 newFac = FactorMax(v, to_max[0])
438                 self.opt_policy[v] = newFac.decision_fun
439                 proj_factors = [fac for fac in proj_factors if fac is not
440                               to_max[0]] + [newFac]
441                 self.display(2, "maximizing", v)
442                 self.display(3, newFac)
443             else:
444                 proj_factors = self.eliminate_var(proj_factors, v)
445         assert len(proj_factors) == 1, "Should there be only one element of
446             proj_factors?"
447         return proj_factors[0].get_value({})
448
449     def show_policy(self):
450         print('\n'.join(df.to_table() for df in self.opt_policy.values()))

```

```

448 class FactorMax(TabFactor):
449     """A factor obtained by maximizing a variable in a factor.
450     Also builds a decision_function. This is based on FactorSum.
451     """
452
453     def __init__(self, dvar, factor):
454         """dvar is a decision variable.
455         factor is a factor that contains dvar and only parents of dvar

```



```

456         """
457         self.dvar = dvar
458         self.factor = factor
459         vars = [v for v in factor.variables if v is not dvar]
460         Factor.__init__(self,vars)
461         self.values = {}
462         self.decision_fun = DecisionFunction(dvar, dvar.parents)
463
464     def get_value(self,assignment):
465         """lazy implementation: if saved, return saved value, else compute
            it"""
466         new_asst = {x:v for (x,v) in assignment.items() if x in
            self.variables}
467         asst = frozenset(new_asst.items())
468         if asst in self.values:
469             return self.values[asst]
470         else:
471             max_val = float("-inf") # -infinity
472             for elt in self.dvar.domain:
473                 fac_val = self.factor.get_value(assignment|{self.dvar:elt})
474                 if fac_val>max_val:
475                     max_val = fac_val
476                     best_elt = elt
477             self.values[asst] = max_val
478             self.decision_fun.assign(assignment, best_elt)
479             return max_val

```

Here are some example queries:

```

decnNetworks.py — (continued)
481 # Example queries:
482 # vf = VE_DN(fire_dn)
483 # vf.optimize()
484 # vf.show_policy()
485
486 # VE_DN.max_display_level = 3 # if you want to show lots of detail
487 # vc = VE_DN(cheating_dn)
488 # vc.optimize()
489 # vc.show_policy()
490
491 def test(dn):
492     rc0dn = RC_DN(dn)
493     rc0v = rc0dn.optimize(algorithm=rc0dn.rc0)
494     rcdn = RC_DN(dn)
495     rcv = rcdn.optimize()
496     assert abs(rc0v-rcv)<1e-10, f"rc0 produces {rc0v}; rc produces {rcv}"
497     vedn = VE_DN(dn)
498     vev = vedn.optimize()
499     assert abs(vev-rcv)<1e-10, f"VE_DN produces {vev}; RC produces {rcv}"
500     print(f"passed unit test. rc0, rc and VE gave same result for {dn}")
501

```

```

502 | if __name__ == "__main__":
503 |     test(fire_dn)

```

12.2 Markov Decision Processes

The following represent a **Markov decision process (MDP)** directly, rather than using the recursive conditioning or variable elimination code.

```

_____mdpProblem.py — Representations for Markov Decision Processes _____
11 | import random
12 | from display import Displayable
13 | from utilities import argmaxd
14 |
15 | class MDP(Displayable):
16 |     """A Markov Decision Process. Must define:
17 |         title a string that gives the title of the MDP
18 |         states the set (or list) of states
19 |         actions the set (or list) of actions
20 |         discount a real-valued discount
21 |     """
22 |
23 |     def __init__(self, title, states, actions, discount, init=0):
24 |         self.title = title
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,
35 |         and other 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 12.29 in Poole and Mackworth [2023]):

```

_____mdpExamples.py — MDP Examples _____
11 | from mdpProblem import MDP, ProblemDomain, distribution
12 | from mdpGUI import GridDomain

```

```

13 import matplotlib.pyplot as plt
14
15 class partyMDP(MDP):
16     """Simple 2-state, 2-Action Partying MDP Example"""
17     def __init__(self, discount=0.9):
18         states = {'healthy', 'sick'}
19         actions = {'relax', 'party'}
20         MDP.__init__(self, "party MDP", states, actions, discount)
21
22     def R(self, s, a):
23         "R(s,a)"
24         return { 'healthy': {'relax': 7, 'party': 10},
25                 'sick': {'relax': 0, 'party': 2 } }[s][a]
26
27     def P(self, s, a):
28         "returns a dictionary of {s1:p1} such that P(s1 | s,a)=p1. Other
29         probabilities are zero."
30         phealthy = { # P('healthy' | s, a)
31                     'healthy': {'relax': 0.95, 'party': 0.7},
32                     'sick': {'relax': 0.5, 'party': 0.1 } }[s][a]
33         return {'healthy': phealthy, 'sick': 1-phealthy}

```

The distribution class is used to represent distributions as they are being created. Probability distributions are represented as *item : value* dictionaries. When being constructed, adding an *item : value* to the dictionary has to act differently when the item is already in the dictionary and when it isn't. The `add_prob` method works whether the item is in the dictionary or not.

mdpProblem.py — (continued)

```

44 class distribution(dict):
45     """A distribution is an item:prob dictionary.
46     Probabilities are added using add_prop.
47     """
48     def __init__(self, d):
49         dict.__init__(self, d)
50
51     def add_prob(self, item, pr):
52         """adds a probability to a distribution.
53         Like dictionary assignment, but if item is already there, the
54         values are summed
55         """
56         if item in self:
57             self[item] += pr
58         else:
59             self[item] = pr
60         return self

```

12.2.1 Problem Domains

An MDP does not contain enough information to simulate a domain, because

- (a) the rewards and resulting state can be correlated (e.g., in the grid domains below, crashing into a wall results in both a negative reward and the agent not moving), and
- (b) it represents the *expected* reward (e.g., a reward of 1 is has the same expected value as a reward of 100 with probability 1/100 and 0 otherwise, but these are different in a simulation).

A problem domain represents a problem as a function result from states and actions into a distribution of (*state, reward*) pairs. This can be a subclass of MDP because it implements R and P. A problem domain also specifies an initial state and coordinate information used by the graphical user interfaces.

```

_____mdpProblem.py — (continued) _____
61 class ProblemDomain(MDP):
62     """A ProblemDomain implements
63     self.result(state, action) -> {(reward, state):probability}.
64     Other pairs have probability are zero.
65     The probabilities must sum to 1.
66     """
67     def __init__(self, title, states, actions, discount,
68                 initial_state=None, x_dim=0, y_dim = 0,
69                 vinit=0, offsets={}):
70         """A problem domain
71         * title is list of titles
72         * states is the list of states
73         * actions is the list of actions
74         * discount is the discount factor
75         * initial_state is the state the agent starts at (for simulation)
76           if known
77         * x_dim and y_dim are the dimensions used by the GUI to show the
78           states in 2-dimensions
79         * vinit is the initial value
80         * offsets is a {action:(x,y)} map which specifies how actions are
81           displayed in GUI
82         """
83         MDP.__init__(self, title, states, actions, discount)
84         if initial_state is not None:
85             self.state = initial_state
86         else:
87             self.state = random.choice(states)
88         self.vinit = vinit # value to reset v,q to
89         # The following are for the GUI:
90         self.x_dim = x_dim
91         self.y_dim = y_dim
92         self.offsets = offsets
93
94     def state2pos(self, state):
95         """When displaying as a grid, this specifies how the state is
96           mapped to (x,y) position.
97         The default is for domains where the (x,y) position is the state

```

```

94         """
95         return state
96
97     def state2goal(self, state):
98         """When displaying as a grid, this specifies how the state is
99         mapped to goal position.
100         The default is for domains where there is no goal
101         """
102         return None
103
104     def pos2state(self, pos):
105         """When displaying as a grid, this specifies how the state is
106         mapped to (x,y) position.
107         The default is for domains where the (x,y) position is the state
108         """
109         return pos
110
111     def P(self, state, action):
112         """Transition probability function
113         returns a dictionary of {s1:p1} such that P(s1 | state,action)=p1.
114         Other probabilities are zero.
115         """
116         res = self.result(state, action)
117         acc = 1e-6 # accuracy for test of equality
118         assert 1-acc<sum(res.values())<1+acc, f"result({state},{action})
119             not a distribution, sum={sum(res.values())}"
120         dist = distribution({})
121         for ((r,s),p) in res.items():
122             dist.add_prob(s,p)
123         return dist
124
125     def R(self, state, action):
126         """Reward function R(s,a)
127         returns the expected reward for doing a in state s.
128         """
129         return sum(r*p for ((r,s),p) in self.result(state, action).items())

```

Tiny Game

The next example is the tiny game from Example 13.1 and Figure 13.1 of Poole and Mackworth [2023], shown here as Figure 12.5. There are 6 states and 4 actions. The state is represented as (x, y) where x counts from zero from the left, and y counts from zero upwards, so the state $(0, 0)$ is on the bottom-left. The actions are upC for up-careful, upR for up-risky, left, and right. Going left from $(0, 2)$ results in a reward of 10 and ending up in state $(0, 0)$; going left from $(0, 1)$ results in a reward of -100 and staying there. Up-risky goes up but with a chance of going left or right. Up careful goes up, but has a reward of -1 . Left and right are deterministic. Crashing into a wall results in a reward of -1 and staying still.

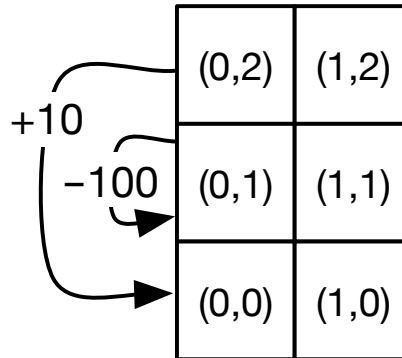


Figure 12.5: Tiny game

(Note that GridDomain means that it can be shown with the MDP GUI in Section 12.2.3).

```

mdpExamples.py — (continued)
34 class MDPTiny(ProblemDomain, GridDomain):
35     def __init__(self, discount=0.9):
36         x_dim = 2 # x-dimension
37         y_dim = 3
38         ProblemDomain.__init__(self,
39             "Tiny MDP", # title
40             [(x,y) for x in range(x_dim) for y in range(y_dim)], #states
41             ['right', 'upC', 'left', 'upR'], #actions
42             discount,
43             x_dim=x_dim, y_dim = y_dim,
44             offsets = {'right':(0.25,0), 'upC':(0,-0.25), 'left':(-0.25,0),
45                       'upR':(0,0.25)}
46         )
47     def result(self, state, action):
48         """return a dictionary of {(r,s):p} where p is the probability of
49           reward r, state s
50           a state is an (x,y) pair
51           """
52         (x,y) = state
53         right = (-x,(1,y)) # reward is -1 if x was 1
54         left = (0,(0,y)) if x==1 else [(-1,(0,0)), (-100,(0,1)),
55                                         (10,(0,0))][y]
56         up = (0,(x,y+1)) if y<2 else (-1,(x,y))
57         if action == 'right':
58             return {right:1}
59         elif action == 'upC':
60             (r,s) = up

```

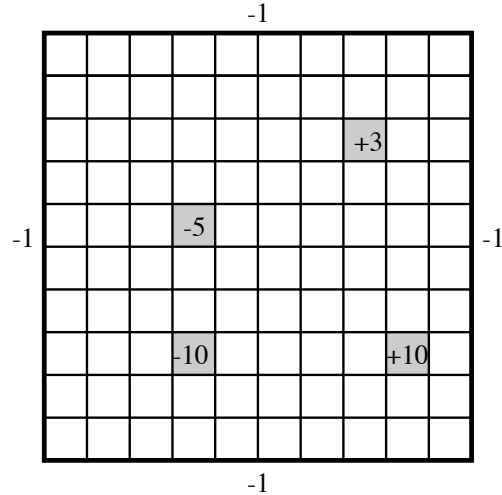


Figure 12.6: Grid world

```

59         return {(r-1,s):1}
60     elif action == 'left':
61         return {left:1}
62     elif action == 'upR':
63         return distribution({left:
64             0.1}).add_prob(right,0.1).add_prob(up,0.8)
65         # Exercise: what is wrong with return {left: 0.1, right:0.1,
66             up:0.8}
67
68 # To show GUI do
69 # MDPTiny().viGUI()

```

Grid World

Here is the domain of Example 12.30 of Poole and Mackworth [2023], shown here in Figure 12.6. A state is represented as (x,y) where x counts from zero from the left, and y counts from zero upwards, so the state $(0,0)$ is on the bottom-left.

```

_____mdpExamples.py — (continued) _____
69 class grid(ProblemDomain, GridDomain):
70     """ x_dim * y_dim grid with rewarding states"""
71     def __init__(self, discount=0.9, x_dim=10, y_dim=10):
72         ProblemDomain.__init__(self,
73             "Grid World",
74             [(x,y) for x in range(y_dim) for y in range(y_dim)], #states
75             ['up', 'down', 'right', 'left'], #actions
76             discount,
77             x_dim = x_dim, y_dim = y_dim,

```

```

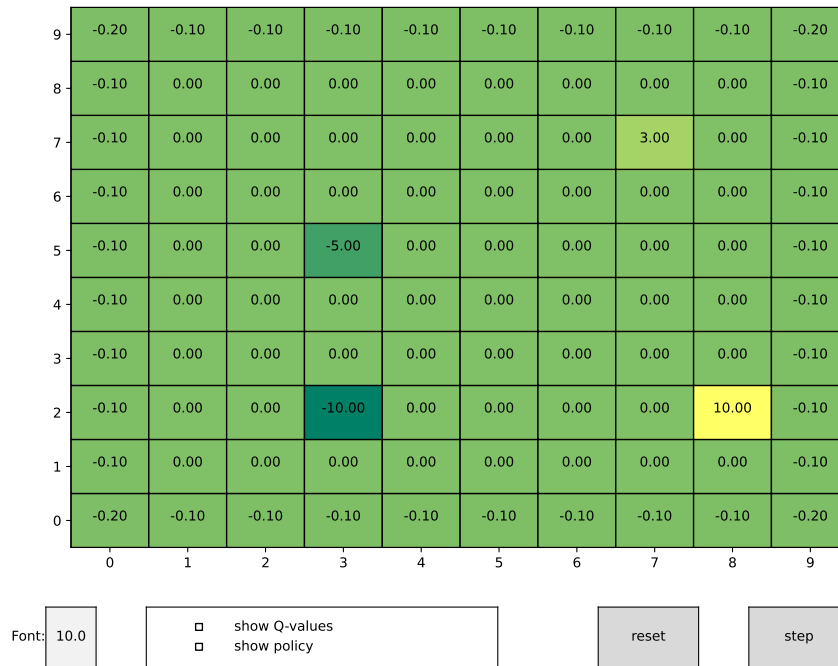
78         offsets = {'right':(0.25,0), 'up':(0,0.25), 'left':(-0.25,0),
79                   'down':(0,-0.25)})
80     self.rewarding_states = {(3,2):-10, (3,5):-5, (8,2):10, (7,7):3 }
81     self.fling_states = {(8,2), (7,7)} # assumed a subset of
82         rewarding_states
83
84     def intended_next(self,s,a):
85         """returns the (reward, state) in the direction a.
86         This is where the agent will end up if to goes in its
87         intended_direction
88         (which it does with probability 0.7).
89         """
90         (x,y) = s
91         if a=='up':
92             return (0, (x,y+1)) if y+1 < self.y_dim else (-1, (x,y))
93         if a=='down':
94             return (0, (x,y-1)) if y > 0 else (-1, (x,y))
95         if a=='right':
96             return (0, (x+1,y)) if x+1 < self.x_dim else (-1, (x,y))
97         if a=='left':
98             return (0, (x-1,y)) if x > 0 else (-1, (x,y))
99
100     def result(self,s,a):
101         """return a dictionary of {(r,s):p} where p is the probability of
102         reward r, state s.
103         a state is an (x,y) pair
104         """
105         r0 = self.rewarding_states[s] if s in self.rewarding_states else 0
106         if s in self.fling_states:
107             return {(r0,(0,0)): 0.25, (r0,(self.x_dim-1,0)):0.25,
108                     (r0,(0,self.y_dim-1)):0.25,
109                     (r0,(self.x_dim-1,self.y_dim-1)):0.25}
110         dist = distribution({})
111         for a1 in self.actions:
112             (r1,s1) = self.intended_next(s,a1)
113             rs = (r1+r0, s1)
114             p = 0.7 if a1==a else 0.1
115             dist.add_prob(rs,p)
116         return dist

```

Figure 12.7 shows the immediate expected reward for each of the 100 states. This was generated using `grid().viGUI()` and carrying out one step.

Monster Game

This is for the game depicted in Figure 12.8 (Example 13.2 of Poole and Mackworth [2023]). There are 25 locations where the agent can be, there can be no prize or there can be a prize in one of the corners ($P_1 \dots P_4$). The agent only gets a positive reward when gets to the prize. The agent can be damaged or undamaged. There are possible monsters at the locations marked with M . If

Figure 12.7: Grid world GUI: `grid().viGUI()`

the agent lands on a monster when it is undamaged, it gets damaged. If the agent lands on a monster when it is damaged, it gets a negative reward. It can get undamaged by going to the location marked with *R*. It gets a negative reward by crashing into a wall. There are $25 * 5 * 2 = 250$ states. There are 4 actions, *up*, *down*, *left*, and *right*; the agent generally goes in the direction of the action, but has a chance of going in one of the other directions.

```

_____mdpExamples.py — (continued)_____
113 class Monster_game(ProblemDomain, GridDomain):
114
115     vwalls = [(0,3), (0,4), (1,4)] # vertical walls right of these locations
116     crash_reward = -1
117
118     prize_locs = [(0,0), (0,4), (4,0), (4,4)]
119     prize_appears_prob = 0.3
120     prize_reward = 10
121
122     monster_locs = [(0,1), (1,1), (2,3), (3,1), (4,2)]
123     monster_appears_prob = 0.4
124     monster_reward_when_damaged = -10
125     repair_stations = [(1,4)]

```

4	P ₁	R			P ₂
3			M		
2					M
1	M	M		M	
0	P ₃				P ₄
	0	1	2	3	4

Figure 12.8: Monster game

```

126
127 def __init__(self, discount=0.9):
128     x_dim = 5
129     y_dim = 5
130     # which damaged and prize to show
131     ProblemDomain.__init__(self,
132         "Monster Game",
133         [(x,y,damaged,prize)
134             for x in range(x_dim)
135             for y in range(y_dim)
136             for damaged in [False,True]
137             for prize in [None]+self.prize_locs], #states
138         ['up', 'down', 'right', 'left'], #actions
139         discount,
140         x_dim = x_dim, y_dim = y_dim,
141         offsets = {'right':(0.25,0), 'up':(0,0.25), 'left':(-0.25,0),
142             'down':(0,-0.25)})
142     self.state = (2,2,False,None)
143
144 def intended_next(self,xy,a):
145     """returns the (reward, (x,y)) in the direction a.
146     This is where the agent will end up if to goes in its
147         intended_direction
148         (which it does with probability 0.7).
149     """
150     (x,y) = xy # original x-y position
151     if a=='up':
152         return (0, (x,y+1)) if y+1 < self.y_dim else
153             (self.crash_reward, (x,y))
154     if a=='down':
155         return (0, (x,y-1)) if y > 0 else (self.crash_reward, (x,y))

```

```

154         if a=='right':
155             if (x,y) in self.vwalls or x+1==self.x_dim: # hit wall
156                 return (self.crash_reward, (x,y))
157             else:
158                 return (0, (x+1,y))
159         if a=='left':
160             if (x-1,y) in self.vwalls or x==0: # hit wall
161                 return (self.crash_reward, (x,y))
162             else:
163                 return (0, (x-1,y))
164
165     def result(self,s,a):
166         """return a dictionary of {(r,s):p} where p is the probability of
167         reward r, state s.
168         a state is an (x,y) pair
169         """
170         (x,y,damaged,prize) = s
171         dist = distribution({})
172         for a1 in self.actions: # possible results
173             mp = 0.7 if a1==a else 0.1
174             mr,(xn,yn) = self.intended_next((x,y),a1)
175             if (xn,yn) in self.monster_locs:
176                 if damaged:
177                     dist.add_prob((mr+self.monster_reward_when_damaged, (xn,yn,True,prize)),
178                                   mp*self.monster_appears_prob)
179                     dist.add_prob((mr, (xn,yn,True,prize)),
180                                   mp*(1-self.monster_appears_prob))
181                 else:
182                     dist.add_prob((mr, (xn,yn,True,prize)),
183                                   mp*self.monster_appears_prob)
184                     dist.add_prob((mr, (xn,yn,False,prize)),
185                                   mp*(1-self.monster_appears_prob))
186             elif (xn,yn) == prize:
187                 dist.add_prob((mr+self.prize_reward, (xn,yn,damaged,None)),
188                               mp)
189             elif (xn,yn) in self.repair_stations:
190                 dist.add_prob((mr, (xn,yn,False,prize)), mp)
191             else:
192                 dist.add_prob((mr, (xn,yn,damaged,prize)), mp)
193         if prize is None:
194             res = distribution({})
195             for (r,(x2,y2,d,p2)),p in dist.items():
196                 res.add_prob((r,(x2,y2,d,None)),
197                               p*(1-self.prize_appears_prob))
198             for pz in self.prize_locs:
199                 res.add_prob((r,(x2,y2,d,pz)),
200                               p*self.prize_appears_prob/len(self.prize_locs))
201         return res
202     else:
203         return dist

```

```

196
197     def state2pos(self, state):
198         """When displaying as a grid, this specifies how the state is
199           mapped to (x,y) position.
200           The default is for domains where the (x,y) position is the state
201           """
202         (x,y,d,p) = state
203         return (x,y)
204
205     def pos2state(self, pos):
206         """When displaying as a grid, this specifies how the state is
207           mapped to (x,y) position.
208           """
209         (x,y) = pos
210         (xs, ys, damaged, prize) = self.state
211         return (x, y, damaged, prize)
212
213     def state2goal(self, state):
214         """the (x,y) position for the goal
215           """
216         (x, y, damaged, prize) = state
217         return prize
218
219 # To see value iterations:
220 # mg = Monster_game()
221 # mg.viGUI() # then run vi a few times
222 # to see other states, exit the GUI
223 # mg.state = (2,2,True,(4,4)) # or other damaged/prize states
224 # mg.viGUI()

```

12.2.2 Value Iteration

The following implements value iteration for Markov decision processes.

A Q function is represented as a dictionary so $Q[s][a]$ is the value for doing action a in state s . The value function is represented as a dictionary so $V[s]$ is the value of state s . Policy π is represented as a dictionary where $\pi[s]$, where s is a state, returns the action.

Note that the following defines `vi` to be a method in MDP.

```

_____mdpProblem.py — (continued) _____
128 def vi(self, n):
129     """carries out n iterations of value iteration, updating value
130       function self.V
131       Returns a Q-function, value function, policy
132       """
133     self.display(3,f"calling vi({n})")
134     for i in range(n):
135         self.Q = {s: {a: self.R(s,a)
136                       +self.discount*sum(p1*self.V[s1]

```

```

136         for (s1,p1) in
            self.P(s,a).items()
137         for a in self.actions}
138         for s in self.states}
139         self.V = {s: max(self.Q[s][a] for a in self.actions)
140         for s in self.states}
141         self.pi = {s: argmaxd(self.Q[s])
142         for s in self.states}
143         return self.Q, self.V, self.pi
144
145 MDP.vi = vi

```

The following shows how this can be used.

```

_____mdpExamples.py — (continued) _____
224 ## Testing value iteration
225 # Try the following:
226 # pt = partyMDP(discount=0.9)
227 # pt.vi(1)
228 # pt.vi(100)
229 # partyMDP(discount=0.99).vi(100)
230 # partyMDP(discount=0.4).vi(100)
231
232 # gr = grid(discount=0.9)
233 # gr.viGUI()
234 # q,v,pi = gr.vi(100)
235 # q[(7,2)]

```

12.2.3 Value Iteration GUI for Grid Domains

A GridDomain is a domain where the states can be mapped into (x,y) positions, and the actions can be mapped into up-down-left-right. They are special because the `viGUI()` method to interact with them. It requires the following values/methods be defined:

- `self.x_dim` and `self.y_dim` define the dimensions of the grid (so the states are (x,y) , where $0 \leq x < \text{self.x_dim}$ and $0 \leq y < \text{self.y_dim}$).
- `self.state2pos(state)` gives the (x,y) position of state. The default is that that states are already (x,y) positions.
- `self.state2goal(state)` gives the (x,y) position of the goal in state. The default is `None`.
- `self.pos2state(pos)` where `pos` is an (x,y) pair, gives the state that is shown at position (x,y) . When the state contain more information than the (x,y) pair, the extra information is taken from `self.state`.
- `self.offsets[a]` defines where to display action `a`, as (x,y) offset for action `a` when displaying Q-values.

```

mdpGUI.py — GUI for value iteration in MDPs
11 import matplotlib.pyplot as plt
12 from matplotlib.widgets import Button, CheckButtons, TextBox
13 from mdpProblem import MDP
14
15 class GridDomain(object):
16
17     def viGUI(self):
18         fig,self.ax = plt.subplots()
19         plt.subplots_adjust(bottom=0.2)
20         stepB = Button(plt.axes([0.8,0.05,0.1,0.075]), "step")
21         stepB.on_clicked(self.on_step)
22         resetB = Button(plt.axes([0.65,0.05,0.1,0.075]), "reset")
23         resetB.on_clicked(self.on_reset)
24         self.qcheck = CheckButtons(plt.axes([0.2,0.05,0.35,0.075]),
25                                     ["show Q-values","show policy"])
26         self.qcheck.on_clicked(self.show_vals)
27         self.font_box = TextBox(plt.axes([0.1,0.05,0.05,0.075]),
28                                 "Font:", textalignment="center")
29         self.font_box.on_submit(self.set_font_size)
30         self.font_box.set_val(str(plt.rcParams['font.size']))
31         self.show_vals(None)
32         plt.show()
33
34     def set_font_size(self, s):
35         plt.rcParams.update({'font.size': eval(s)})
36         plt.draw()
37
38     def show_vals(self,event):
39         self.ax.cla() # clear the axes
40
41         array = [[self.V[self.pos2state((x,y))]] for x in range(self.x_dim)]
42                 for y in range(self.y_dim)]
43         self.ax.pcolormesh([x-0.5 for x in range(self.x_dim+1)],
44                             [y-0.5 for y in range(self.y_dim+1)],
45                             array, edgecolors='black',cmap='summer')
46
47         # for cmap see
48         # https://matplotlib.org/stable/tutorials/colors/colormaps.html
49         if self.qcheck.get_status()[1]: # "show policy"
50             for x in range(self.x_dim):
51                 for y in range(self.y_dim):
52                     state = self.pos2state((x,y))
53                     maxv = max(self.Q[state][a] for a in self.actions)
54                     for a in self.actions:
55                         if self.Q[state][a] == maxv:
56                             # draw arrow in appropriate direction
57                             xoff, yoff = self.offsets[a]
58                             self.ax.arrow(x,y,xoff*2,yoff*2,
59                                             color='red',width=0.05, head_width=0.2,
60                                             length_includes_head=True)

```

```

59         if self.qcheck.get_status()[0]: # "show q-values"
60             self.show_q(event)
61         else:
62             self.show_v(event)
63             self.ax.set_xticks(range(self.x_dim))
64             self.ax.set_xticklabels(range(self.x_dim))
65             self.ax.set_yticks(range(self.y_dim))
66             self.ax.set_yticklabels(range(self.y_dim))
67             plt.draw()
68
69     def on_step(self, event):
70         self.step()
71         self.show_vals(event)
72
73     def step(self):
74         """The default step is one step of value iteration"""
75         self.vi(1)
76
77     def show_v(self, event):
78         """show values"""
79         for x in range(self.x_dim):
80             for y in range(self.y_dim):
81                 state = self.pos2state((x,y))
82                 self.ax.text(x,y,"{val:.2f}".format(val=self.V[state]),ha='center')
83
84     def show_q(self, event):
85         """show q-values"""
86         for x in range(self.x_dim):
87             for y in range(self.y_dim):
88                 state = self.pos2state((x,y))
89                 for a in self.actions:
90                     xoff, yoff = self.offsets[a]
91                     self.ax.text(x+xoff,y+yoff,
92                                 "{val:.2f}".format(val=self.Q[state][a]),ha='center')
93
94     def on_reset(self, event):
95         self.V = {s:self.vinit for s in self.states}
96         self.Q = {s: {a: self.vinit for a in self.actions} for s in
97                     self.states}
98         self.show_vals(event)
99
100 # to use the GUI do some of:
101 # python -i mdpExamples.py
102 # MDPtiny(discount=0.9).viGUI()
103 # grid(discount=0.9).viGUI()
104 # Monster_game(discount=0.9).viGUI()

```

Figure 12.9 shows the user interface for the tiny domain, which can be obtained using

```
MDPtiny(discount=0.9).viGUI()
```

resizing it, checking “show q-values” and “show policy”, and clicking “step” a

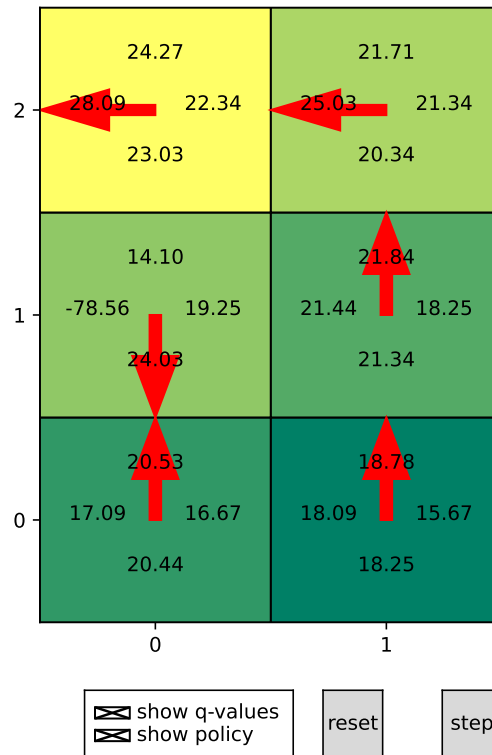


Figure 12.9: Interface for tiny example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The left-most number is for the left action; the rightmost number is for the right action; the uppermost is for the *upR* (up-risky) action and the lowest number is for the *upC* action. The arrow points to the action(s) with the maximum Q-value. Use `MDPtiny().viGUI()` after loading `mdpExamples.py`

few times.

```
To run the demo in class do:
% python -i mdpExamples.py
MDPtiny(discount=0.9).viGUI()
```

Figure 12.10 shows the user interface for the grid domain, which can be obtained using `grid(discount=0.9).viGUI()` resizing it, checking “show q-values” and “show policy”, and clicking “step” a few times.

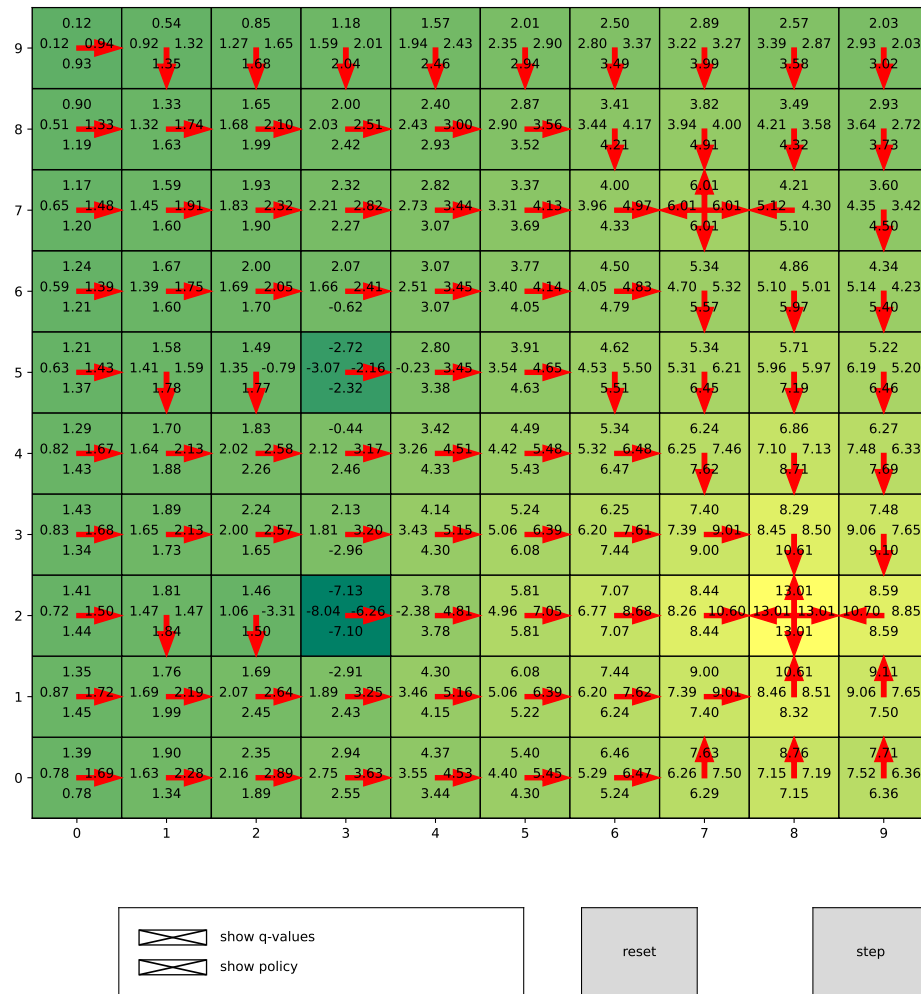


Figure 12.10: Interface for grid example, after a number of steps. Each rectangle represents a state. In each rectangle are the 4 Q-values for the state. The leftmost number is for the left action; the rightmost number is for the right action; the uppermost is for the up action and the lowest number is for the down action. The arrow points to the action(s) with the maximum Q-value. From `grid(discount=0.9).viGUI()`

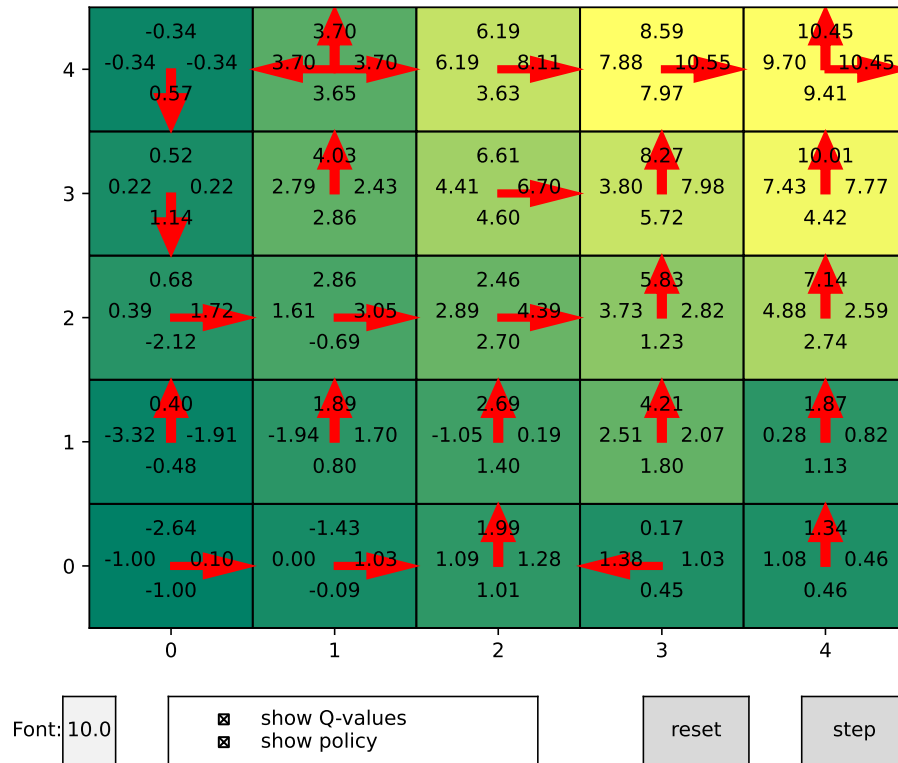


Figure 12.11: Q-values and optimal policy for the monster game, for the states where the agent is damaged and the goal is in the top-right.

Figure 12.11 shows the optimal policy and Q-values after convergence (clicking “step” more does not change the Q-values) for the states where the agent is damaged and the goal is in the top-right. There are 10 times as many states as positions, so we can’t show them all. See the commented out lines at the end of the Monster game code to reproduce this figure.

Exercise 12.1 Computing q before v may seem like a waste of space because we don’t need to store q in order to compute the value function or the policy. Change the algorithm so that it loops through the states and actions once per iteration, and only stores the value function and the policy. Note that to get the same results as before, you would need to make sure that you use the previous value of v in the computation not the current value of v . Does using the current value of v hurt the algorithm or make it better (in approaching the actual value function)?

12.2.4 Asynchronous Value Iteration

This implements asynchronous value iteration, storing Q .

A Q function is represented using $Q[s][a]$ as the value for doing action with a in state s .

```

_____mdpProblem.py — (continued)_____
147 def avi(self,n):
148     states = list(self.states)
149     actions = list(self.actions)
150     for i in range(n):
151         s = random.choice(states)
152         a = random.choice(actions)
153         self.Q[s][a] = (self.R(s,a) + self.discount *
154                         sum(p1 * max(self.Q[s1][a1]
155                                     for a1 in self.actions)
156                             for (s1,p1) in self.P(s,a).items()))
157     return self.Q
158
159 # make this a method for the MPD class:
160 MDP.avi = avi

```

The following shows how `avi` can be used.

```

_____mdpExamples.py — (continued)_____
238 ## Testing asynchronous value iteration
239 # Try the following:
240 # pt = partyMDP(discount=0.9)
241 # pt.avi(10)
242 # pt.vi(1000)
243
244 # gr = grid(discount=0.9)
245 # q = gr.avi(100000)
246 # q[(7,2)]
247
248 def test_MDP(mdp, discount=0.9, eps=0.01):
249     """tests vi and avi give the same answer for a MDP class mdp
250     """
251     mdp1 = mdp(discount=discount)
252     q1,v1,pi1 = mdp1.vi(100)
253     mdp2 = mdp(discount=discount)
254     q2 = mdp2.avi(1000)
255     same = all(abs(q1[s][a]-q2[s][a]) < eps
256                for s in mdp1.states
257                for a in mdp1.actions)
258     assert same, "vi and avi are different:\n{q1}\n{q2}"
259     print(f"passed unit test. vi and avi gave same result for {mdp1.title}")
260
261 if __name__ == "__main__":
262     test_MDP(partyMDP)

```

Exercise 12.2 Implement value iteration that stores the V -values rather than the Q -values. Does it work better than storing Q ? (What might “better” mean?)

Exercise 12.3 In asynchronous value iteration, try a number of different ways to choose the states and actions to update (e.g., sweeping through the state-action pairs, choosing them at random). Note that the best way may be to determine which states have had their Q -values changed the most, and then update the previous ones, but that is not so straightforward to implement, because you need to find those previous states.

Reinforcement Learning

13.1 Representing Agents and Environments

The reinforcement learning agents and environments are instances of the general agent architecture of Section 2.1, where the percepts are (reward, state) pairs. The state here is the state of the environment, not the state of the agent. Thus this is assuming that the environment is **fully observable**.

Agents are told what actions are available to it to use, but don't initially know anything about the possible states.

- An agent implements the method `select_action` takes a (reward, state) returns the next action (and updates the state of the agent).
- An environment implements the method `do` that takes an action and returns a (reward, state) pair.

These are alternated to simulate the system. The simulation starts with the agent choosing the initial action given the state, using the method `initial_action(state)`, which typically remembers the state and returns a random action.

13.1.1 Environments

RL environments have names to make tracing easier. An environment also has a list of all of the actions that can be carried out in the environment. It is initialized with the initial state.

```
_____rlProblem.py — Representations for Reinforcement Learning _____  
11 | import random  
12 | import math  
13 | from display import Displayable
```

```

14 from agents import Agent, Environment
15 from utilities import select_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
20            state"""
21         self.name = name          # the name of the environment
22         self.actions = actions    # list of all actions
23         self.state = state        # initial state
24         self.reward = None        # last reward
25
26     # must implement do(action)->(reward,state)

```

13.1.2 Agents

An agent initially knows what actions it can carry out (its abilities). The interactions is started by calling `initial_action`, which tells the agent what the initial state is. An agent typically remembers the state and returns an action. It has no reason to prefer one action over another, so it chooses an action at random.

```

rlProblem.py — (continued)
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

```

At each time step, an agent selects its next action given the reward it received and the environment.

```

rlProblem.py — (continued)
43     def select_action(self, reward, state):
44         """
45         Select the action given the reward and state
46         Remember the action in self.action
47         This implements "Act randomly" and should be overridden!
48         """

```

```

49         self.reward = reward
50         self.action = random.choice(self.actions)
51         return self.action
52
53     def v(self, state):
54         """estimate of the value of doing a best action in state.
55         """
56         return max(self.q(state,a) for a in self.actions)
57
58     def q(self, state, action):
59         """estimate of value of doing action in state. Should be
60         overirdden to be useful.
61         """
62         return 0

```

13.1.3 Simulating an Environment-Agent Interaction

The interaction between an agent and an environment is mediated by a simulator that calls the agent and the environment in turn. Simulate in this section is similar to Simulate of Section 2.1, except it is initialized by `agent.initial_action(state)`, and the rewards are accumulated.

```

_____rlProblem.py — (continued)_____
63 import matplotlib.pyplot as plt
64
65 class Simulate(Displayable):
66     """simulate the interaction between the agent and the environment
67     for n time steps.
68     Returns a pair of the agent state and the environment state.
69     """
70     def __init__(self, agent, environment):
71         self.agent = agent
72         self.env = environment
73         self.reward_history = [] # for plotting
74         self.step = 0
75         self.sum_rewards = 0
76
77     def start(self):
78         self.action = self.agent.initial_action(self.env.state)
79         return self
80
81     def go(self, n):
82         for i in range(n):
83             self.step += 1
84             (reward,state) = self.env.do(self.action)
85             self.display(2,f"step={self.step} reward={reward},
86                         state={state}")
87             self.sum_rewards += reward
88             self.reward_history.append(reward)

```

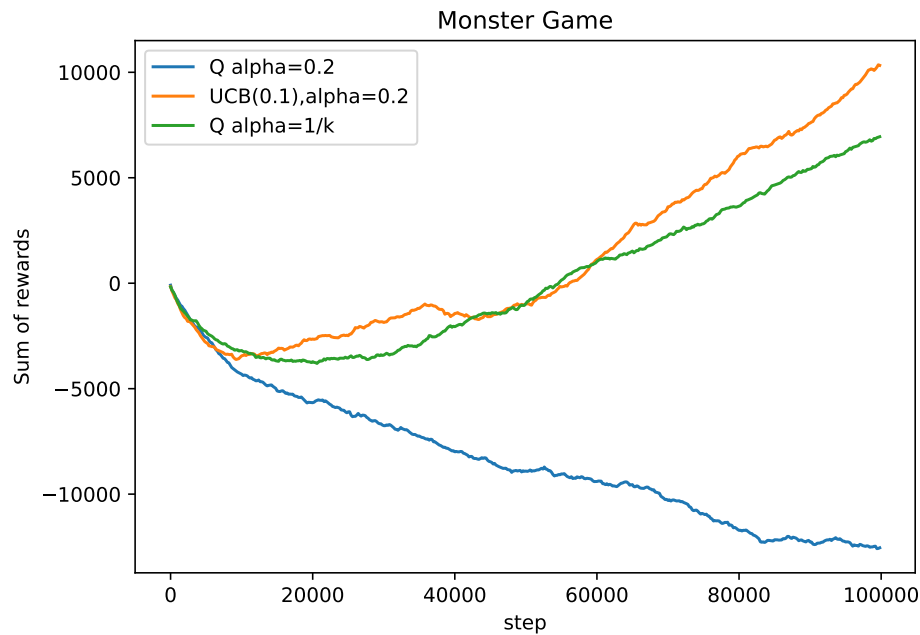


Figure 13.1: Plotting the performance of some algorithms for the monster game

```

88         self.action = self.agent.select_action(reward, state)
89         self.display(2, f"    action={self.action}")
90     return self

```

The following plots the sum of rewards as a function of the step in a simulation. Figure 13.1 shows the performance of three algorithms for the Monster Game (Sections 12.2.1 and 13.1.6). On the x-axis is the number of actions. On the y-axis is the cumulative reward. The algorithm corresponding to the blue line has not learned very well; the plot keeps going down (but less than it did initially). The learner represented by the green line starts getting positive performance after about 20,000 steps. It took about 55,000 steps for it to have gained back the cost of exploration (when it crosses $y = 0$). The learner represented by the orange line seems to have learned quicker, but is more erratic. Each algorithm should be run multiple times, because the performance can vary a lot, even for the same problem, algorithm, and parameter settings. This graph can be reproduced (but the lines will be different) using code at the bottom of `RLQlearner.py`.

```

rlProblem.py — (continued)
91     def plot(self, label=None, step_size=None, xscale='linear'):
92         """
93         plots the rewards history in the simulation
94         label is the label for the plot
95         step_size is the number of steps between each point plotted
96         xscale is 'log' or 'linear'

```



```

97         returns sum of rewards
98         """
99
100        if step_size is None: #for long simulations (> 999), only plot some
            points
101            step_size = max(1,len(self.reward_history)//500)
102        if label is None:
103            label = self.agent.name
104        plt.ion()
105        plt.xscale(xscale)
106        plt.title(self.env.name)
107        plt.xlabel("step")
108        plt.ylabel("Sum of rewards")
109        sum_history, sum_rewards = acc_rews(self.reward_history, step_size)
110        plt.plot(range(0,len(self.reward_history),step_size), sum_history,
            label=label)
111        plt.legend()
112        plt.draw()
113        return sum_rewards
114
115    def acc_rews(rews,step_size):
116        """returns the rolling sum of the values, sampled each step_size, and
            the sum
117        """
118        acc = []
119        sumr = 0; i=0
120        for e in rews:
121            sumr += e
122            i += 1
123            if (i%step_size == 0): acc.append(sumr)
124        return acc, sumr

```

13.1.4 Party Environment

Here is the definition of the simple 2-state, 2-action decision about whether to party or relax (Example 12.29 in Poole and Mackworth [2023]). (Compare to the MDP representation of page 298)

```

_____rlExamples.py — Some example reinforcement learning environments_____
11 from rlProblem import RL_env
12 class Party_env(RL_env):
13     def __init__(self):
14         RL_env.__init__(self, "Party Decision", ["party", "relax"],
            "healthy")
15
16     def do(self, action):
17         """updates the state based on the agent doing action.
18         returns reward,state
19         """
20         if self.state=="healthy":

```

```

21         if action=="party":
22             self.state = "healthy" if flip(0.7) else "sick"
23             self.reward = 10
24         else: # action=="relax"
25             self.state = "healthy" if flip(0.95) else "sick"
26             self.reward = 7
27     else: # self.state=="sick"
28         if action=="party":
29             self.state = "healthy" if flip(0.1) else "sick"
30             self.reward = 2
31         else:
32             self.state = "healthy" if flip(0.5) else "sick"
33             self.reward = 0
34     return self.reward, self.state

```

13.1.5 Environment from a Problem Domain

Env_fom_ProblemDomain takes a ProblemDomain (page 299) and constructs an environment that can be used for reinforcement learners.

As explained in Section 12.2.1, the representation of an MDP does not contain enough information to simulate a system, because it loses any dependency between the rewards and the resulting state (e.g., hitting the wall and having a negative reward may be correlated), and only represents the expected value of rewards, not how they are distributed. The ProblemDomain class defines the result method to map states and actions into distributions over (reward, state) pairs.

```

rlProblem.py — (continued)
126
127 class Env_from_ProblemDomain(RL_env):
128     def __init__(self, prob_dom):
129         RL_env.__init__(self, prob_dom.title, prob_dom.actions,
130                         prob_dom.state)
131         self.problem_domain = prob_dom
132         self.state = prob_dom.state
133         self.x_dim = prob_dom.x_dim
134         self.y_dim = prob_dom.y_dim
135         self.offsets = prob_dom.offsets
136         self.state2pos = self.problem_domain.state2pos
137         self.state2goal = self.problem_domain.state2goal
138         self.pos2state = self.problem_domain.pos2state
139
140     def do(self, action):
141         """updates the state based on the agent doing action.
142         returns state,reward
143         """
144         (self.reward, self.state) =
145             select_from_dist(self.problem_domain.result(self.state, action))
146         self.problem_domain.state = self.state

```

4	P ₁	R			P ₂
3			M		
2					M
1	M	M		M	
0	P ₃				P ₄
	0	1	2	3	4

Figure 13.2: Monster game

```

145     self.display(2,f"do({action} -> ({self.reward}, {self.state}))"
146     return (self.reward,self.state)

```

13.1.6 Monster Game Environment

This is for the game depicted in Figure 13.2 (Example 13.2 of Poole and Mackworth [2023]). This is an alternative representation to that of Section 12.2.1, which defined the distribution over reward-state pairs. This directly builds a simulator, which might be easier to understand and easier adapt to new environments.

There are $25 * 5 * 2 = 250$ states. The agent does not know anything about how the environment works; it just knows what actions are available to it and what state it is in. It has to learn what to do.

```

rlExamples.py — (continued) —
36 import random
37 from utilities import flip
38 from rlProblem import RL_env
39
40 class Monster_game_env(RL_env):
41     x_dim = 5
42     y_dim = 5
43
44     vwalls = [(0,3), (0,4), (1,4)] # vertical walls right of these locations
45     hwalls = [] # not implemented
46     crashed_reward = -1
47
48     prize_locs = [(0,0), (0,4), (4,0), (4,4)]
49     prize_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         self.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.x_dim-1 or (self.x,self.y) in self.vwalls:
93                 self.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                 self.reward += self.crashed_reward
99             else:

```

```

98         self.x += -1
99     elif actual_direction == "up":
100         if self.y==self.y_dim-1:
101             self.reward += self.crashed_reward
102         else:
103             self.y += 1
104     elif actual_direction == "down":
105         if self.y==0:
106             self.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             self.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         self.reward += self.prize_reward
125         self.prize = None
126
127     # Statistics
128     self.number_steps += 1
129     self.accumulated_rewards += self.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
134         self.reward>self.accumulated_rewards:
135         self.zero_crossing = self.number_steps
136     self.display(2,"",self.number_steps,self.accumulated_rewards,
137                 self.accumulated_rewards/self.number_steps,sep="\t")

```

The following methods are used by the GUI (Section 13.7, page 345) so that the states can be shown.

```

139     """ For GUI
140     def state2pos(self,state):
141         """the (x,y) position for the state
142         """

```

```

143         (x, y, damaged, prize) = state
144         return (x,y)
145
146     def state2goal(self,state):
147         """the (x,y) position for the goal
148         """
149         (x, y, damaged, prize) = state
150         return prize
151
152     def pos2state(self,pos):
153         """the state corresponding to the (x,y) position.
154         The damages and prize are not shown in the GUI
155         """
156         (x,y) = pos
157         return (x, y, self.damaged, self.prize)

```

13.2 Q Learning

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

```

rlQLearner.py — Q Learning
11 import random
12 import math
13 from display import Displayable
14 from utilities import argmaxe, argmaxd, flip
15 from rlProblem import RL_agent, epsilon_greedy, ucb
16
17 class Q_learner(RL_agent):
18     """A Q-learning agent has
19     belief-state consisting of
20     state is the previous state (initialized by RL_agent
21     q is a {(state,action):value} dict
22     visits is a {(state,action):n} dict. n is how many times action was
23     done in state
24     acc_rewards is the accumulated reward
25     """
26
27 rlQLearner.py — (continued)
28
29 def __init__(self, name, actions, discount,
30             exploration_strategy=epsilon_greedy, es_kwargs={},
31             alpha_fun=lambda _:0.2, Qinit=0):
32     """
33     name is string representation of the agent
34     actions is the set of actions the agent can do
35     discount is the discount factor

```

```

33     exploration_strategy is the exploration function, default
34         "epsilon_greedy"
35     es_kwargs is extra arguments of exploration_strategy
36     alpha_fun is a function that computes alpha from the number of
37         visits
38     Qinit is the initial q-value
39     """
40     RL_agent.__init__(self, actions)
41     self.name = name
42     self.discount = discount
43     self.exploration_strategy = exploration_strategy
44     self.es_kwargs = es_kwargs
45     self.alpha_fun = alpha_fun
46     self.Qinit = Qinit
47     self.acc_rewards = 0
48     self.Q = {}
49     self.visits = {}

```

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

```

                                     rlQLearner.py — (continued)
49     def initial_action(self, state):
50         """ Returns the initial action; selected at random
51         Initialize Data Structures
52         """
53         self.state = state
54         self.Q[state] = {act:self.Qinit for act in self.actions}
55         self.visits[state] = {act:0 for act in self.actions}
56         self.action = self.exploration_strategy(state, self.Q[state],
57             self.visits[state],**self.es_kwargs)
58         self.display(2, f"Initial State: {state} Action {self.action}")
59         self.display(2, "s\\ta\\tr\\ts\\tQ")
60         # display looks best if states and actions are < 8 characters
61         return self.action
62
63     def select_action(self, reward, next_state):
64         """give reward and next state, select next action to be carried
65         out"""
66         if next_state not in self.visits: # next_state not seen before
67             self.Q[next_state] = {act:self.Qinit for act in self.actions}
68             self.visits[next_state] = {act:0 for act in self.actions}
69         self.visits[self.state][self.action] +=1
70         alpha = self.alpha_fun(self.visits[self.state][self.action])
71         self.Q[self.state][self.action] += alpha*(
72             reward
73             + self.discount * max(self.Q[next_state].values())
74             - self.Q[self.state][self.action])
75         self.display(2,self.state, self.action, reward, next_state,
76             self.Q[self.state][self.action], sep='\\t')

```

```

76         self.action = self.exploration_strategy(next_state,
77             self.Q[next_state],
78             self.visits[next_state],**self.es_kwargs)
79         self.state = next_state
80         self.display(3,f"Agent {self.name} doing {self.action} in state
            {self.state}")
81         return self.action

```

The GUI requires the $q(s,a)$ functions:

```

rlQLearner.py — (continued)
82 def q(self,s,a):
83     if s in self.Q and a in self.Q[s]:
84         return self.Q[s][a]
85     else:
86         return self.Qinit

```

SARSA is the same as Q-learning except in the action selection. SARSA changes 3 lines:

```

rlQLearner.py — (continued)
88 class SARSA(Q_learner):
89     def __init__(self,*args, **nargs):
90         Q_learner.__init__(self,*args, **nargs)
91
92     def select_action(self, reward, next_state):
93         """give reward and next state, select next action to be carried
94            out"""
95         if next_state not in self.visits: # next state not seen before
96             self.Q[next_state] = {act:self.Qinit for act in self.actions}
97             self.visits[next_state] = {act:0 for act in self.actions}
98         self.visits[self.state][self.action] +=1
99         alpha = self.alpha_fun(self.visits[self.state][self.action])
100         next_action = self.exploration_strategy(next_state,
101             self.Q[next_state],
102             self.visits[next_state],**self.es_kwargs)
103         self.Q[self.state][self.action] += alpha*(
104             reward
105             + self.discount * self.Q[next_state][next_action]
106             - self.Q[self.state][self.action])
107         self.display(2,self.state, self.action, reward, next_state,
108             self.Q[self.state][self.action], sep='\t')
109         self.state = next_state
110         self.action = next_action
111         self.display(3,f"Agent {self.name} doing {self.action} in state
            {self.state}")
112         return self.action

```


13.2.1 Exploration Strategies

Two explorations strategies are defined: epsilon-greedy and upper confidence bound (UCB).

In general an exploration strategy takes two arguments, and some optional arguments depending on the strategy.

- *State* is the state that action is chosen for
- *Qs* is a $\{action : q_value\}$ dictionary for the state
- *visits* is a $\{action : n\}$ dictionary for the current state; where n is the number of times that the action has been carried out in the current state.

```

rlProblem.py — (continued)
148 def epsilon_greedy(state, Qs, visits={}, epsilon=0.2):
149     """select action given epsilon greedy
150     Qs is the {action:Q-value} dictionary for current state
151     visits is ignored
152     epsilon is the probability of acting randomly
153     """
154     if flip(epsilon):
155         return random.choice(list(Qs.keys())) # act randomly
156     else:
157         return argmaxd(Qs) # pick an action with max Q
158
159 def ucb(state, Qs, visits, c=1.4):
160     """select action given upper-confidence bound
161     Qs is the {action:Q-value} dictionary for current state
162     visits is the {action:n} dictionary for current state
163
164     0.01 is to prevent divide-by zero when visits[a]==0
165     """
166     Ns = sum(visits.values())
167     ucb1 = {a:Qs[a]+c*math.sqrt(Ns/(0.01+visits[a]))
168             for a in Qs.keys()}
169     action = argmaxd(ucb1)
170     return action

```

Exercise 13.1 Implement a soft-max action selection. Choose a temperature that works well for the domain. Explain how you picked this temperature. Compare the epsilon-greedy, ucb, soft-max and optimism in the face of uncertainty for various parameter settings.

13.2.2 Testing Q-learning

The unit tests are for the 2-action 2-state decision about whether to relax or party (Example 12.29 of Poole and Mackworth [2023]).

Note that simulating the same agent multiple times does not restart the agent; it keeps learning. Try the plotting some of the other methods; make sure to try multiple agents with the same parameter values before deciding whether a method with particular parameter settings is good or not. To do this, make sure you construct a new agent.

```

112 ##### TEST CASES #####
113 from rlProblem import Simulate, epsilon_greedy, ucb, Env_from_ProblemDomain
114 from rlExamples import Party_env, Monster_game_env
115 from rlQLearner import Q_learner
116 from mdpExamples import MDPTiny, partyMDP
117
118 def test_RL(learnerClass, mdp=partyMDP, env=Party_env(), discount=0.9,
119            eps=2, **kwargs):
120     """tests whether RL on env has the same (within eps) Q-values as vi on
121     mdp"""
122     mdp1 = mdp(discount=discount)
123     q1, v1, pi1 = mdp1.vi(1000)
124     ag = learnerClass(learnerClass.__name__, env.actions, discount,
125                      **kwargs)
126     sim = Simulate(ag, env).start()
127     sim.go(100000)
128     same = all(abs(ag.q(s,a)-q1[s][a]) < eps
129               for s in mdp1.states
130               for a in mdp1.actions)
131     assert same, (f"Unit test failed for {env.name}, in {ag.name} Q="
132                  +str({(s,a):ag.q(s,a) for s in mdp1.states
133                       for a in mdp1.actions})
134                  +f" in vi Q={q1}")
135     print(f"Unit test passed. For {env.name}, {ag.name} has same Q-value as
136           value iteration")
137 if __name__ == "__main__":
138     test_RL(Q_learner, alpha_fun=lambda k:10/(9+k))
139     #test_RL(SARSA) # should this pass? Why or why not?

```

The following are some calls you can play with. Run the commented-out code. Try other agents, including agents with the same settings.

```

137 #env = Party_env()
138 env = Env_from_ProblemDomain(MDPTiny())
139 # Some RL agents with different parameters:
140 ag = Q_learner("eps (0.1) greedy", env.actions, 0.7)
141 ag_ucb = Q_learner("ucb", env.actions, 0.7, exploration_strategy = ucb,
142                  es_kwargs={'c':0.1})
143 ag_opt = Q_learner("optimistic", env.actions, 0.7, Qinit=100,
144                  es_kwargs={'epsilon':0})
145 ag_exp_m = Q_learner("more explore", env.actions, 0.7,
146                  es_kwargs={'epsilon':0.5})
147 ag_greedy = Q_learner("disc 0.1", env.actions, 0.1, Qinit=100)

```

```

145 sa = SARSA("SARSA", env.actions, 0.9)
146 suchb = SARSA("SARSA ucb", env.actions, 0.9, exploration_strategy = ucb,
    es_kwargs={'c':1})
147
148 sim_ag = Simulate(ag,env).start()
149
150 # sim_ag.go(1000)
151 # ag.Q # get the learned Q-values
152 # sim_ag.plot()
153 # sim_uchb = Simulate(ag_uchb,env).start(); sim_uchb.go(1000); sim_uchb.plot()
154 # Simulate(ag_opt,env).start().go(1000).plot()
155 # Simulate(ag_exp_m,env).start().go(1000).plot()
156 # Simulate(ag_greedy,env).start().go(1000).plot()
157 # Simulate(sa,env).start().go(1000).plot()
158 # Simulate(suchb,env).start().go(1000).plot()
159
160 from mdpExamples import MDPTiny
161 envt = Env_from_ProblemDomain(MDPTiny())
162 agt = Q_learner("Q alpha=0.8", envt.actions, 0.8)
163 #Simulate(agt, envt).start().go(1000).plot()
164
165 ##### Monster Game #####
166 mon_env = Monster_game_env()
167 mag1 = Q_learner("Q alpha=0.2", mon_env.actions, 0.9)
168 #Simulate(mag1,mon_env).start().go(100000).plot()
169 mag_uchb = Q_learner("UCB(0.1),alpha=0.2", mon_env.actions, 0.9,
    exploration_strategy = ucb, es_kwargs={'c':0.1})
170 #Simulate(mag_uchb,mon_env).start().go(100000).plot()
171
172
173 mag2 = Q_learner("Q alpha=1/k", mon_env.actions, 0.9,
    alpha_fun=lambda k:1/k)
174 #Simulate(mag2,mon_env).start().go(100000).plot()
175 mag3 = Q_learner("alpha=10/(9+k)", mon_env.actions, 0.9,
    alpha_fun=lambda k:10/(9+k))
176 #Simulate(mag3,mon_env).start().go(100000).plot()
177
178
179 mag4 = Q_learner("uchb & alpha=10/(9+k)", mon_env.actions, 0.9,
    alpha_fun=lambda k:10/(9+k),
    exploration_strategy = ucb, es_kwargs={'c':0.1})
180 #Simulate(mag4,mon_env).start().go(100000).plot()
181
182
183

```

13.3 Q-learning with Experience Replay

A bounded buffer remembers values up to size `buffer_size`. Random values can be obtained using `get`. Once the bounded buffer is full, all old experiences have the same chance of being in the buffer.

rlQExperienceReplay.py — Q-Learner with Experience Replay

```
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, new_value):
22         if self.number_added < self.buffer_size:
23             self.buffer[self.number_added] = new_value
24         else:
25             if flip(self.buffer_size/self.number_added):
26                 position = random.randrange(self.buffer_size)
27                 self.buffer[position] = new_value
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, name, actions, discount,
35                 max_buffer_size=10000,
36                 num_updates_per_action=10, burn_in=100, **q_kwargs):
37         """Q-learner with experience replay
38         name is the name of the agent (e.g., in a game)
39         actions is the set of actions the agent can do
40         discount is the discount factor
41         max_buffer_size is the maximum number of past experiences that is
42             remembered
43         burn_in is the number of steps before using old experiences
44         num_updates_per_action is the number of q-updates for past
45             experiences per action
46         q_kwargs are any extra parameters for Q_learner
47         """
48         Q_learner.__init__(self, name, actions, discount, **q_kwargs)
49         self.experience_buffer = BoundedBuffer(max_buffer_size)
50         self.num_updates_per_action = num_updates_per_action
51         self.burn_in = burn_in
52
53     def select_action(self, reward, next_state):
54         """give reward and new state, select next action to be carried
55             out"""
56         self.experience_buffer.add((self.state, self.action, reward, next_state))
57         #remember experience

```

```

54         if next_state not in self.visits: # next_state not seen before
55             self.Q[next_state] = {act:self.Qinit for act in self.actions}
56             self.visits[next_state] = {act:0 for act in self.actions}
57         self.visits[self.state][self.action] +=1
58         alpha = self.alpha_fun(self.visits[self.state][self.action])
59         self.Q[self.state][self.action] += alpha*(
60             reward
61             + self.discount * max(self.Q[next_state].values())
62             - self.Q[self.state][self.action])
63         self.display(2,self.state, self.action, reward, next_state,
64                     self.Q[self.state][self.action], sep='\t')
65         # do some updates from experience buffer
66         if self.experience_buffer.number_added > self.burn_in:
67             for i in range(self.num_updates_per_action):
68                 (s,a,r,ns) = self.experience_buffer.get()
69                 self.visits[s][a] +=1 # is this correct?
70                 alpha = self.alpha_fun(self.visits[s][a])
71                 self.Q[s][a] += alpha * (r +
72                                         self.discount* max(self.Q[ns][na]
73                                                             for na in self.actions)
74                                         -self.Q[s][a] )
75         ### CHOOSE NEXT ACTION ###
76         self.action = self.exploration_strategy(next_state,
77                                                 self.Q[next_state],
78                                                 self.visits[next_state],**self.es_kwargs)
79         self.state = next_state
80         self.display(3,f"Agent {self.name} doing {self.action} in state
81                     {self.state}")
82         return self.action

```

The following code plots the performance. The experience replay learner performance cannot be directly compared to Q-learning as it does more updates per action.

```

_____rlQExperienceReplay.py — (continued) _____
82 from rlProblem import Simulate
83 from rlExamples import Monster_game_env
84 from rlQLearner import mag1, mag2, mag3
85
86 mon_env = Monster_game_env()
87 mag1ar = Q_ER_learner("Q_ER", mon_env.actions,0.9,
88                      num_updates_per_action=5, burn_in=100)
89 # Simulate(mag1ar,mon_env).start().go(100000).plot()
90
91 mag3ar = Q_ER_learner("Q_ER alpha=10/(9+k)", mon_env.actions, 0.9,
92                      num_updates_per_action=50, burn_in=1000,
93                      alpha_fun=lambda k:10/(9+k))
94 # Simulate(mag3ar,mon_env).start().go(100000).plot()
95
96 from rlQLearner import test_RL
97 if __name__ == "__main__":

```

```
98 | test_RL(Q_ER_learner, alpha_fun=lambda k:10/(9+k))
```

Exercise 13.2 Why does this have a burn-in? What problem might this solve? How much does the burn-in affect the result?

Exercise 13.3 What is a fair way to compare the learning rate of Q_ER_learner and Q_learner, or Q_ER_learners with different values of num_updates_per_action? (Would this matter if the environment is a simulation versus in the real world?) Implement a comparison that counts the number of updates, rather than the number of actions. How much does num_updates_per_action matter?

13.4 Stochastic Policy Learning Agent

The following agent is like a Q-learning agent but maintains a stochastic policy. The policy is represented as unnormalized counts for each action in a state (as in a Dirichlet distribution). This is the code described in Section 14.7.2 and Figure 14.10 of Poole and Mackworth [2023].

```

_____rlStochasticPolicy.py — Simulations of agents learning _____
11 from display import Displayable
12 import utilities # argmaxall for (element,value) pairs
13 import matplotlib.pyplot as plt
14 import random
15 from rlQLearner import Q_learner
16
17 class StochasticPIAgent(Q_learner):
18     """This agent maintains the Q-function for each state.
19     Chooses the best action using empirical distribution over actions
20     """
21     def __init__(self, name, actions, discount=0, pi_init=1, **nargs):
22         """
23         name is the name of the agent (e.g., in a game)
24         actions is the set of actions the agent can do.
25         discount is the discount factor (0 is appropriate if there is a
26             single state)
27         pi_init gives the prior counts (Dirichlet prior) for the policy
28             (must be >0)
29         """
30         #self.max_display_level = 3
31         Q_learner.__init__(self, name, actions, discount,
32                             exploration_strategy=self.action_from_stochastic_policy,
33                             **nargs)
34         self.pi_init = pi_init
35         self.pi = {}
36
37     def initial_action(self, state):
38         """ update policy pi then do initial action from Q_learner
39         """
40         self.pi[state] = {act:self.pi_init for act in self.actions}

```

```

39         return Q_learner.initial_action(self, state)
40
41     def action_from_stochastic_policy(self, next_state, qs, vs):
42         a_best = utilities.argmaxd(self.Q[self.state])
43         self.pi[self.state][a_best] += 1
44         if next_state not in self.pi:
45             self.pi[next_state] = {act:self.pi_init for act in
46                                     self.actions}
47         return select_from_dist(self.pi[next_state])
48
49     def normalize(dist):
50         """dict is a {value:number} dictionary, where the numbers are all
51         non-negative
52         returns dict where the numbers sum to one
53         """
54         tot = sum(dist.values())
55         return {var:val/tot for (var,val) in dist.items()}
56
57     def select_from_dist(dist):
58         rand = random.random()
59         for (act,prob) in normalize(dist).items():
60             rand -= prob
61             if rand < 0:
62                 return act

```

The agent can be tested on the reinforcement learning benchmarks:

```

rlStochasticPolicy.py — (continued)
62 ##### Testing on RL benchmarks #####
63 from rlProblem import Simulate
64 import rlExamples
65 mon_env = rlExamples.Monster_game_env()
66 magspi = StochasticPIAgent(mon_env.name, mon_env.actions,0.9)
67 #Simulate(magspi,mon_env).start().go(100000).plot()
68 magspi10 = StochasticPIAgent("stoch 10/(9+k)", mon_env.actions,0.9,
69                             alpha_fun=lambda k:10/(9+k))
70 #Simulate(magspi10,mon_env).start().go(100000).plot()
71
72 from rlQLearner import test_RL
73 if __name__ == "__main__":
74     test_RL(StochasticPIAgent, alpha_fun=lambda k:10/(9+k))

```

Exercise 13.4 Test some other ways to determine the probabilities for the stochastic policy in StochasticPIAgent. (It currently can be seen as using a Dirichlet where the probability represents the proportion of times each action is best plus pseudo-counts).

Replace `self.pi[self.state][a_best] += 1` with something like `self.pi[self.state][a_best] *= c` for some $c > 1$. E.g., $c = 1.1$ so it chooses that action 10% more, independently of the number of times tried. (Try to change the code as little as possible; make it so that either the original or different values of c can be run without changing your code. Warning: watch out for overflow.)

- (a) Try for multiple c ; which one works best for the Monster game?
- (b) Suggest an alternative way to update the probabilities in the policy (e.g., adding δ to policy that is then normalized or some other methods). How well does it work?

13.5 Model-based Reinforcement Learner

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

A model-based reinforcement learner builds a Markov decision process model of the domain, simultaneously learns the model and plans with that model.

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

- $Q[s][a]$ is dictionary that, given state s and action a returns the Q -value, the estimate of the future (discounted) value of being in state s and doing action a . (Note that Q is the list but q is the function.)
- $R[s][a]$ is dictionary that, given a (s, a) state s and action a is the average reward received from doing a in state s .
- $T[s][a][s']$ is dictionary that, given states s and s' and action a returns the number of times a was done in state s and the result was state s' . Note that s' is only a key if it has been the result of doing a in s ; there are no zero counts recorded.
- $visits[s][a]$ is dictionary that, given state s and action a returns the number of times action a was carried out in state s . This is the C of Figure 13.6 of Poole and Mackworth [2023].

Note that $visits[s][a] = \sum_{s'} T[s][a][s']$ but is stored separately to keep the code more readable.

The main difference to Figure 13.6 of Poole and Mackworth [2023] is the code below does a fixed number of asynchronous value iteration updates per step.

```

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, name, actions, discount,
```



```

21         exploration_strategy=epsilon_greedy, es_kwargs={},
22         Qinit=0,
23         updates_per_step=10):
24     """name is the name 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.name = name
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

```

rlModelLearner.py — (continued)

```

41 def initial_action(self, state):
42     """ Returns the initial action; selected at random
43     Initialize Data Structures
44
45     """
46     self.action = RL_agent.initial_action(self, state)
47     self.T = {self.state: {a: {} for a in self.actions}}
48     self.visits = {self.state: {a: 0 for a in self.actions}}
49     self.Q = {self.state: {a: self.Qinit for a in self.actions}}
50     self.R = {self.state: {a: 0 for a in self.actions}}
51     self.states_list = [self.state] # list of states encountered
52     self.display(2, f"Initial State: {state} Action {self.action}")
53     self.display(2, "s\ta\tr\ts\tQ")
54     return self.action

```

rlModelLearner.py — (continued)

```

56 def select_action(self, reward, next_state):
57     """do num_steps of interaction with the environment
58     for each action, do updates_per_step iterations of asynchronous
59     value iteration
60
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

```

```

68     else:
69         self.T[self.state][self.action][next_state] = 1
70         self.visits[self.state][self.action] += 1
71         self.R[self.state][self.action] +=
72             (reward-self.R[self.state][self.action])/self.visits[self.state][self.action]
73         st,act = self.state,self.action #initial state-action pair for AVI
74         for update in range(self.updates_per_step):
75             self.Q[st][act] = self.R[st][act]+self.discount*(
76                 sum(self.T[st][act][nst]/self.visits[st][act]*self.v(nst)
77                     for nst in self.T[st][act].keys()))
78             st = random.choice(self.states_list)
79             act = random.choice(self.actions)
80             self.state = next_state
81             self.action = self.exploration_strategy(next_state,
82                 self.Q[next_state],
83                 self.visits[next_state],**self.es_kwargs)
84         return self.action
85
86     def q(self, state, action):
87         if state in self.Q and action in self.Q[state]:
88             return self.Q[state][action]
89         else:
90             return self.Qinit

```

rlModelLearner.py — (continued)

```

90 from rlExamples import Monster_game_env
91 mon_env = Monster_game_env()
92 mb11 = Model_based_reinforcement_learner("model-based(1)",
93     mon_env.actions, 0.9, updates_per_step=1)
94 # Simulate(mb11,mon_env).start().go(100000).plot()
95 mb110 = Model_based_reinforcement_learner("model-based(10)",
96     mon_env.actions, 0.9, updates_per_step=10)
97 # Simulate(mb110,mon_env).start().go(100000).plot()
98
99 from rlGUI import rlGUI
100 #gui = rlGUI(mon_env, mb11)
101
102 from rlQLearner import test_RL
103 if __name__ == "__main__":
104     test_RL(Model_based_reinforcement_learner)

```

Exercise 13.5 If there were only one update per step, the algorithm could be made simpler and use less space. Explain how. Does it make it more efficient? Is it worthwhile having more than one update per step for the games implemented here?

Exercise 13.6 It is possible to implement the model-based reinforcement learner by replacing Q , R , T , $visits$, res_states with a single dictionary that, given a state and action returns a tuple corresponding to these data structures. Does this make the algorithm easier to understand? Does this make the algorithm more efficient?

Exercise 13.7 If the states and the actions were mapped into integers, the dictionaries could be implemented perhaps more efficiently as arrays. How would the code need to change? Implement this for the monster game. Is it more efficient?

Exercise 13.8 In `random_choice` in the updates of `select_action`, all state-action pairs have the same chance of being chosen. Does selecting state-action pairs proportionally to the number of times visited work better than what is implemented? Provide evidence for your answer.

13.6 Reinforcement Learning with Features

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

This section covers Q-learning with features, where the Q-function is a linear function of feature values.

13.6.1 Representing Features

A feature is a real-valued function from state and action. For an environment, you construct a function that takes a state and an action and returns a list (vector) of real numbers.

This code only does feature engineering: the feature set is redesigned for each problem. Deep RL uses deep learning to learn features, turns out to be trickier to get to work than is generally assumed.

`party_features3` and `party_features4` return lists of feature values for the party decision. `party_features4` has one extra feature.

```

_____rlGameFeature.py — Feature-based Reinforcement Learner _____
11 from rlExamples import Monster_game_env
12 from rlProblem import RL_env
13
14 def party_features3(state,action):
15     return [1, state=="sick", action=="party"]
16
17 def party_features4(state,action):
18     return [1, state=="sick", action=="party", state=="sick" and
        action=="party"]

```

Exercise 13.9 With `party_features3` what policies can be discovered? What policies cannot be represented as

The `monster_features` defines the vector of feature values for the given state and action.

```

_____rlGameFeature.py — (continued) _____
20 def monster_features(state,action):

```

```

21     """returns the list of feature values for the state-action pair
22     """
23     assert action in Monster_game_env.actions, f"Monster game, unknown
        action: {action}"
24     (x,y,d,p) = state
25     # f1: would go to a monster
26     f1 = monster_ahead(x,y,action)
27     # f2: would crash into wall
28     f2 = wall_ahead(x,y,action)
29     # f3: action is towards a prize
30     f3 = towards_prize(x,y,action,p)
31     # f4: damaged and action is toward repair station
32     f4 = towards_repair(x,y,action) if d else 0
33     # f5: damaged and towards monster
34     f5 = 1 if d and f1 else 0
35     # f6: damaged
36     f6 = 1 if d else 0
37     # f7: not damaged
38     f7 = 1-f6
39     # f8: damaged and prize ahead
40     f8 = 1 if d and f3 else 0
41     # f9: not damaged and prize ahead
42     f9 = 1 if not d and f3 else 0
43     features = [1,f1,f2,f3,f4,f5,f6,f7,f8,f9]
44     # the next 20 features are for 5 prize locations
45     # and 4 distances from outside in all directions
46     for pr in Monster_game_env.prize_locs+[None]:
47         if p==pr:
48             features += [x, 4-x, y, 4-y]
49         else:
50             features += [0, 0, 0, 0]
51     # fp04 feature for y when prize is at 0,4
52     # this knows about the wall to the right of the prize
53     if p==(0,4):
54         if x==0:
55             fp04 = y
56         elif y<3:
57             fp04 = y
58         else:
59             fp04 = 4-y
60     else:
61         fp04 = 0
62     features.append(fp04)
63     return features
64
65 def monster_ahead(x,y,action):
66     """returns 1 if the location expected to get to by doing
67     action from (x,y) can contain a monster.
68     """
69     if action == "right" and (x+1,y) in Monster_game_env.monster_locs:

```

```

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

```

```

118         return 1
119     else:
120         return 0
121
122 def towards_repair(x,y,action):
123     """returns 1 if action is towards the repair station.
124     """
125     if action == "up" and (x>0 and y<4 or x==0 and y<2):
126         return 1
127     elif action == "left" and x>1:
128         return 1
129     elif action == "right" and x==0 and y<3:
130         return 1
131     elif action == "down" and x==0 and y>2:
132         return 1
133     else:
134         return 0

```

The following uses a simpler set of features. In particular, it only considers whether the action will most likely result in a monster position or a wall, and whether the action moves towards the current prize.

```

_____rlGameFeature.py — (continued)_____
136 def simp_features(state,action):
137     """returns a list of feature values for the state-action pair
138     """
139     assert action in Monster_game_env.actions
140     (x,y,d,p) = state
141     # f1: would go to a monster
142     f1 = monster_ahead(x,y,action)
143     # f2: would crash into wall
144     f2 = wall_ahead(x,y,action)
145     # f3: action is towards a prize
146     f3 = towards_prize(x,y,action,p)
147     return [1,f1,f2,f3]

```

13.6.2 Feature-based RL learner

This learns a linear function approximation of the Q-values. It requires the function *get_features* that given a state and an action returns a list of values for all of the features. Each environment requires this function to be provided.

```

_____rlFeatures.py — Feature-based Reinforcement Learner_____
11 import random
12 from rlProblem import RL_agent, epsilon_greedy, ucb
13 from display import Displayable
14 from utilities import argmaxe, flip
15 import rlGameFeature
16
17 class SARSA_LFA_learner(RL_agent):

```

```

18     """A SARSA with linear function approximation (LFA) learning agent has
19     """
20     def __init__(self, name, actions, discount,
21                  get_features=rlGameFeature.party_features4,
22                  exploration_strategy=epsilon_greedy, es_kwargs={},
23                  step_size=0.01, winit=0):
24         """name is the name of the agent (e.g., in a game)
25         actions is the set of actions the agent can do
26         discount is the discount factor
27         get_features is a function get_features(state,action) -> list of
28             feature values
29         exploration_strategy is the exploration function, default
30             "epsilon_greedy"
31         es_kwargs is extra keyword arguments of the exploration_strategy
32         step_size is gradient descent step size
33         winit is the initial value of the weights
34         """
35         RL_agent.__init__(self, actions)
36         self.name = name
37         self.discount = discount
38         self.exploration_strategy = exploration_strategy
39         self.es_kwargs = es_kwargs
40         self.get_features = get_features
41         self.step_size = step_size
42         self.winit = winit

```

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

```

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

```

do takes in the number of steps.

```

_____rlFeatures.py — (continued)_____
52
53     def q(self, state, action):
54         """returns Q-value of the state and action for current weights
55         """
56         return dot_product(self.weights, self.get_features(state, action))
57
58     def select_action(self, reward, next_state):
59         """do num_steps of interaction with the environment"""

```

```

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

```

Test code:

```

rlFeatures.py — (continued)
86 from rlProblem import Simulate
87 from rlExamples import Party_env, Monster_game_env
88 import rlGameFeature
89 from rlGUI import rlGUI
90
91 party = Party_env()
92 pa3 = SARSA_LFA_learner(party.name, party.actions, 0.9,
93                        rlGameFeature.party_features3)
94 # Simulate(pa3,party).start().go(300).plot()
95 pa4 = SARSA_LFA_learner(party.name, party.actions, 0.9,
96                        rlGameFeature.party_features4)
97 # Simulate(pa4,party).start().go(300).plot()
98
99 mon_env = Monster_game_env()
100 fa1 = SARSA_LFA_learner("LFA", mon_env.actions, 0.9,
101                        rlGameFeature.monster_features)
102 # Simulate(fa1,mon_env).start().go(100000).plot()
103 fas1 = SARSA_LFA_learner("LFA (simp features)", mon_env.actions, 0.9,
104                        rlGameFeature.simp_features)
105 #Simulate(fas1,mon_env).start().go(100000).plot()

```



```

102 # rlGUI(mon_env, SARSA_LFA_learner(mon_env.name, mon_env.actions, 0.9,
    rlGameFeature.monster_features))
103
104 from rlQLearner import test_RL
105 if __name__ == "__main__":
106     test_RL(SARSA_LFA_learner, es_kwargs={'epsilon':1}) # random exploration

```

Exercise 13.10 How does the step-size affect performance? Try different step sizes (e.g., 0.1, 0.001, other sizes in-between). Explain the behavior you observe. Which step size works best for this example. Explain what evidence you are basing your prediction on.

Exercise 13.11 Does having extra features always help? Does it sometime help? Does whether it helps depend on the step size? Give evidence for your claims.

Exercise 13.12 For each of the following first predict, then plot, then explain the behavior you observed:

- (a) SARSA_LFA, Model-based learning (with 1 update per step) and Q-learning for 10,000 steps 20% exploring followed by 10,000 steps 100% exploiting
- (b) SARSA_LFA, model-based learning and Q-learning for
 - i) 100,000 steps 20% exploring followed by 100,000 steps 100% exploit
 - ii) 10,000 steps 20% exploring followed by 190,000 steps 100% exploit
- (c) Suppose your goal was to have the best accumulated reward after 200,000 steps. You are allowed to change the exploration rate at a fixed number of steps. For each of the methods, which is the best position to start exploiting more? Which method is better? What if you wanted to have the best reward after 10,000 or 1,000 steps?

Based on this evidence, explain when it is preferable to use SARSA_LFA, Model-based learner, or Q-learning.

Important: you need to run each algorithm more than once. Your explanation should include the variability as well as the typical behavior.

Exercise 13.13 In the call to `self.exploration_strategy`, what should the counts be? (The code above will fail for `ucb`, for example.) Think about the case where there are too many states. Suppose we are just learning for a neighborhood of a current state (e.g., a fixed number of steps away the from the current state); how could the algorithm be modifies to make sure it has at least explored the close neighborhood of the current state?

13.7 GUI for RL

This implements an an interactive graphical user interface for reinforcement learners. It lets the uses choose the actions and visualize the value function and/or the Q-function.

Figure 13.3 shows the GUI for the tiny game (see commented out code at the end of the file) after a 18 actions by the user. The 6 states are shown in a

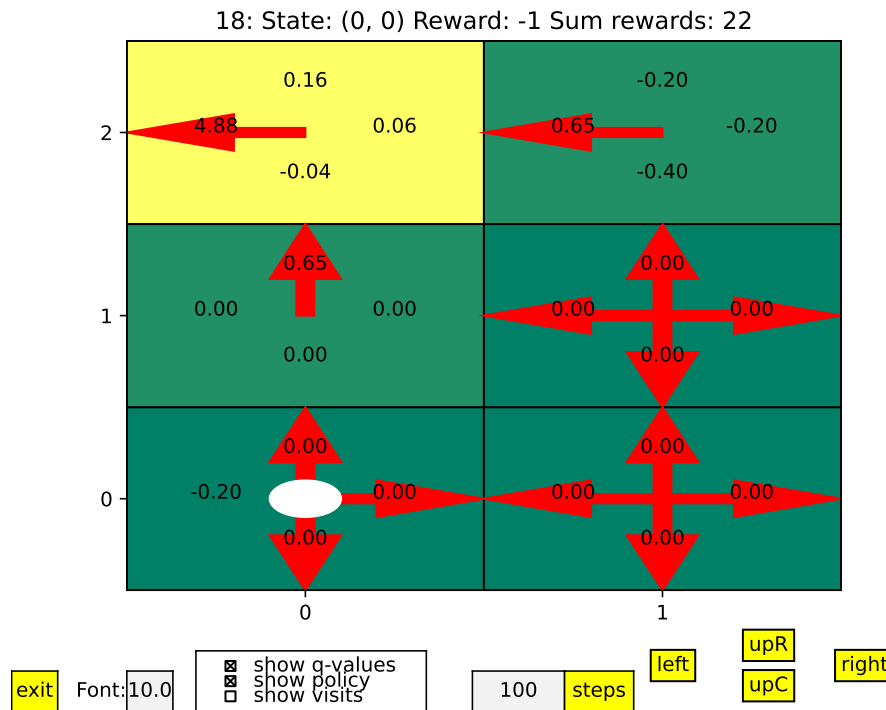


Figure 13.3: Graphical User Interface for tiny game

grid; each rectangle is a state. Within each state are 4 numbers, corresponding to the 4 actions, that give the Q-value for that state and action. The red arrows correspond to the actions with maximal Q-value for each state. The 4 yellow buttons are arranged in the same order as the Q-values. The white ellipse shows the current position of the agent. The user can simulate the agent by clicking on one of these actions. They can also click on “steps” to simulate 100 steps (in this case). The check-boxes are used to show the q-values, the policy (the red arrows) and the visits – the number of times each action has been carried out in each state (when q-values is not checked). When neither q-values or visits is checked the value for the state is shown.

Figure 13.4 shows the GUI for the monster game after 1000 steps. From the top line, you can see the agent is at location (3,3) – shown by the white dot – is damaged and the goal is at (0,4) – shown by the green dot. It is instructive to try to control the agent by clicking on the actions on the bottom left: it only does what is expected 70% of the time.

Warning: Exit is not working, because it is only interrupting one thread.

```

11 | import matplotlib.pyplot as plt
    | _____rGUI.py — Reinforcement Learning GUI_____

```

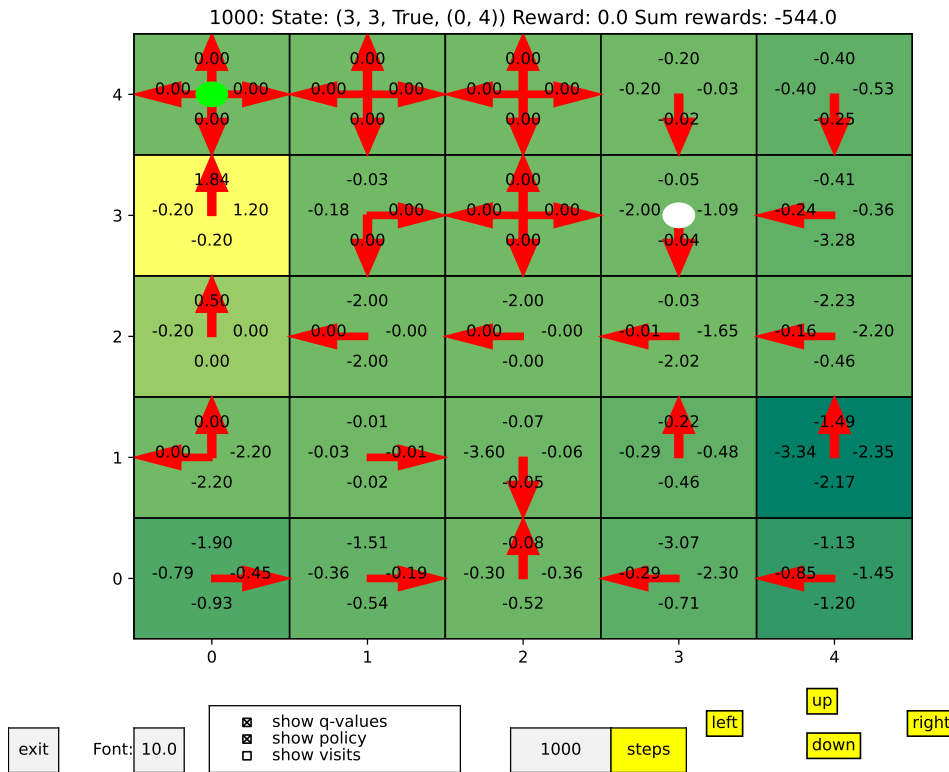


Figure 13.4: Graphical User Interface for Monster game

```

12 from matplotlib.widgets import Button, CheckButtons, TextBox
13 from rlProblem import Simulate
14
15 class rlGUI(object):
16     def __init__(self, env, agent):
17         """
18         """
19         self.env = env
20         self.agent = agent
21         self.state = self.env.state
22         self.x_dim = env.x_dim
23         self.y_dim = env.y_dim
24         if 'offsets' in vars(env): # 'offsets' is defined in environment
25             self.offsets = env.offsets
26         else: # should be more general
27             self.offsets = {'right':(0.25,0), 'up':(0,0.25),
28                             'left':(-0.25,0), 'down':(0,-0.25)}
29         # replace the exploration strategy with GUI

```

```

29     self.orig_exp_strategy = self.agent.exploration_strategy
30     self.agent.exploration_strategy = self.actionFromGUI
31     self.do_steps = 0
32     self.quit = False
33     self.action = None
34
35     def go(self):
36         self.q = self.agent.q
37         self.v = self.agent.v
38         try:
39             self.fig, self.ax = plt.subplots()
40             plt.subplots_adjust(bottom=0.2)
41             self.actButtons =
42                 {self.fig.text(0.8+self.offsets[a][0]*0.4, 0.1+self.offsets[a][1]*0.1, a,
43                     bbox={'boxstyle': 'square', 'color': 'yellow', 'ec': 'black'},
44                     picker=True): a #, fontsize=fontsize): a
45             for a in self.env.actions:
46                 self.fig.canvas.mpl_connect('pick_event', self.sel_action)
47                 self.sim = Simulate(self.agent, self.env)
48                 self.show()
49                 self.sim.start()
50                 self.sim.go(1000000000000) # go forever
51         except ExitGUI:
52             plt.close()
53
54     def show(self):
55         #plt.ion() # interactive (why doesn't this work?)
56         self.qcheck = CheckButtons(plt.axes([0.2, 0.05, 0.25, 0.075]),
57             ["show q-values", "show policy", "show
58                 visits"])
59         self.qcheck.on_clicked(self.show_vals)
60         self.font_box = TextBox(plt.axes([0.125, 0.05, 0.05, 0.05]), "Font:",
61             textalignment="center")
62         self.font_box.on_submit(self.set_font_size)
63         self.font_box.set_val(str(plt.rcParams['font.size']))
64         self.step_box = TextBox(plt.axes([0.5, 0.05, 0.1, 0.05]), "",
65             textalignment="center")
66         self.step_box.set_val("100")
67         self.stepsButton = Button(plt.axes([0.6, 0.05, 0.075, 0.05]), "steps",
68             color='yellow')
69         self.stepsButton.on_clicked(self.steps)
70         self.exitButton = Button(plt.axes([0.0, 0.05, 0.05, 0.05]), "exit",
71             color='yellow')
72         self.exitButton.on_clicked(self.exit)
73         self.show_vals(None)
74
75     def set_font_size(self, s):
76         plt.rcParams.update({'font.size': eval(s)})
77         plt.draw()

```

```

73     def exit(self, s):
74         self.quit = True
75         raise ExitGUI
76
77     def show_vals(self, event):
78         self.ax.cla()
79         self.ax.set_title(f"{self.sim.step}: State: {self.state} Reward:
80             {self.env.reward} Sum rewards: {self.sim.sum_rewards}")
81         array = [[self.v(self.env.pos2state((x,y))) for x in
82             range(self.x_dim)
83                 for y in range(self.y_dim)]
84             self.ax.pcolormesh([x-0.5 for x in range(self.x_dim+1)],
85                 [x-0.5 for x in range(self.y_dim+1)],
86                 array, edgecolors='black', cmap='summer')
87
88         # for cmap see
89         https://matplotlib.org/stable/tutorials/colors/colormaps.html
90     if self.qcheck.get_status()[1]: # "show policy"
91         for x in range(self.x_dim):
92             for y in range(self.y_dim):
93                 state = self.env.pos2state((x,y))
94                 maxv = max(self.agent.q(state,a) for a in
95                     self.env.actions)
96                 for a in self.env.actions:
97                     xoff, yoff = self.offsets[a]
98                     if self.agent.q(state,a) == maxv:
99                         # draw arrow in appropriate direction
100                         self.ax.arrow(x,y,xoff*2,yoff*2,
101                             color='red',width=0.05, head_width=0.2,
102                             length_includes_head=True)
103
104     if goal := self.env.state2goal(self.state):
105         self.ax.add_patch(plt.Circle(goal, 0.1, color='lime'))
106     self.ax.add_patch(plt.Circle(self.env.state2pos(self.state), 0.1,
107         color='w'))
108     if self.qcheck.get_status()[0]: # "show q-values"
109         self.show_q(event)
110     elif self.qcheck.get_status()[2] and 'visits' in vars(self.agent):
111         # "show visits"
112         self.show_visits(event)
113     else:
114         self.show_v(event)
115         self.ax.set_xticks(range(self.x_dim))
116         self.ax.set_xticklabels(range(self.x_dim))
117         self.ax.set_yticks(range(self.y_dim))
118         self.ax.set_yticklabels(range(self.y_dim))
119         plt.draw()
120
121     def sel_action(self, event):
122         self.action = self.actButtons[event.artist]

```

```

116 def show_v(self,event):
117     """show values"""
118     for x in range(self.x_dim):
119         for y in range(self.y_dim):
120             state = self.env.pos2state((x,y))
121             self.ax.text(x,y,"{val:.2f}".format(val=self.agent.v(state)),ha='center')
122
123 def show_q(self,event):
124     """show q-values"""
125     for x in range(self.x_dim):
126         for y in range(self.y_dim):
127             state = self.env.pos2state((x,y))
128             for a in self.env.actions:
129                 xoff, yoff = self.offsets[a]
130                 self.ax.text(x+xoff,y+yoff,
131                             "{val:.2f}".format(val=self.agent.q(state,a)),ha='center')
132
133 def show_visits(self,event):
134     """show q-values"""
135     for x in range(self.x_dim):
136         for y in range(self.y_dim):
137             state = self.env.pos2state((x,y))
138             for a in self.env.actions:
139                 xoff, yoff = self.offsets[a]
140                 if state in self.agent.visits and a in
141                     self.agent.visits[state]:
142                     num_visits = self.agent.visits[state][a]
143                 else:
144                     num_visits = 0
145                 self.ax.text(x+xoff,y+yoff,
146                             str(num_visits),ha='center')
147
148 def steps(self,event):
149     """do the steps given in step box"""
150     num_steps = int(self.step_box.text)
151     if num_steps > 0:
152         self.do_steps = num_steps-1
153         self.action = self.action_from_orig_exp_strategy()
154
155 def action_from_orig_exp_strategy(self):
156     """returns the action from the original explorations strategy"""
157     visits = self.agent.visits[self.state] if 'visits' in
158         vars(self.agent) else {}
159     return
160         self.orig_exp_strategy(self.state,{a:self.agent.q(self.state,a)
161             for a in self.agent.actions},
162                                 visits,**self.agent.es_kwargs)
163
164 def actionFromGUI(self, state, *args, **kwargs):
165     """called as the exploration strategy by the RL agent.

```

```

162         returns an action, either from the GUI or the original exploration
           strategy
163         """
164         self.state = state
165         if self.do_steps > 0: # use the original
166             self.do_steps -= 1
167             return self.action_from_orig_exp_strategy()
168         else: # get action from the user
169             self.show_vals(None)
170             while self.action == None and not self.quit: #wait for user
171                 action
172                 plt.pause(0.05) # controls reaction time of GUI
173             act = self.action
174             self.action = None
175             return act
176
177     class ExitGUI(Exception):
178         """Exit does nothing.
179         """
180         pass
181
182     from rlExamples import Monster_game_env
183     from mdpExamples import MDptiny, Monster_game
184     from rlQLearner import Q_learner, SARSA
185     from rlStochasticPolicy import StochasticPIAgent
186     from rlProblem import Env_from_ProblemDomain, epsilon_greedy, ucb
187
188     # Choose an Environment
189     env = Env_from_ProblemDomain(MDptiny())
190     # env = Env_from_ProblemDomain(Monster_game())
191     # env = Monster_game_env()
192
193     # Choose an algorithm
194     # gui = rlGUI(env, Q_learner("Q", env.actions, 0.9)); gui.go()
195     # gui = rlGUI(env, SARSA("SARSA", env.actions, 0.9)); gui.go()
196     # gui = rlGUI(env, SARSA("SARSA alpha(k)=k:10/(9+k)", env.actions, 0.9,
197         alpha_fun=lambda k:10/(9+k))); gui.go()
198     # gui = rlGUI(env, SARSA("SARSA-UCB", env.actions, 0.9,
199         exploration_strategy = ucb, es_kwargs={'c':0.1})); gui.go()
200     # gui = rlGUI(env, StochasticPIAgent("Q", env.actions, 0.9,
201         alpha_fun=lambda k:10/(9+k))); gui.go()

```


Multiagent Systems

This chapter considers searching game trees and reinforcement learning for games.

14.1 Minimax

The following code implements search for two-player, zero-sum, perfect-information (fully-observable) games. One player only wins when another player loses. Such games can be modeled with

- a single value (utility) which one agent (the maximizing agent) is trying to maximize and the other agent (the minimizing agent) is trying to minimize
- a game tree where the nodes correspond to state of the game (or the history of moves)
- each node is labelled by the player who controls the next move (the maximizing player or the minimizing player)
- the children of non-terminal node correspond to all of the actions by the agent controlling the node
- nodes at the end of the game have no children and are labeled with the value of the node (e.g., +1 for win, 0 for tie, -1 for loss).

The aim of the minimax searcher is, given a state, to find the optimal (maximizing or minimizing depending on the agent) move.

14.1.1 Creating a two-player game

```

masProblem.py — A Multiagent Problem
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 the node 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 if this is a leaf node"""
29         return self.allchildren is None
30
31     def children(self):
32         """returns the list of all children."""
33         return self.allchildren
34
35     def evaluate(self):
36         """returns the evaluation for this node if it is a leaf"""
37         return self.value
38
39     def __repr__(self):
40         return self.name

```

The following gives the tree of Figure 14.1 (Figure 11.5 of Poole and Mackworth [2023]); only the leaf nodes are part of the tree; the other values are described Poole and Mackworth [2023, Section 14.3.1]. 888 is used as a value for those nodes without a value in the tree. (If you look at the trace of alpha-beta pruning, 888 never appears).

```

masProblem.py — (continued)
41 fig10_5 = Node("a", True, None, [
42     Node("b", False, None, [
43         Node("d", True, None, [
44             Node("h", False, None, [
45                 Node("h1", True, 7, None),
46                 Node("h2", True, 9, None)]],
47         Node("i", False, None, [
48             Node("i1", True, 6, None),
49             Node("i2", True, 888, None)]])],

```

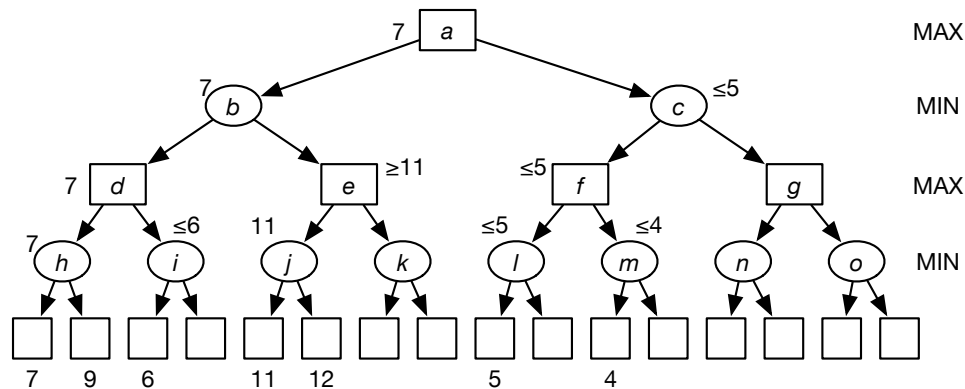


Figure 14.1: Example search tree

```

50     Node("e", True, None, [
51         Node("j", False, None, [
52             Node("j1", True, 11, None),
53             Node("j2", True, 12, None)]),
54         Node("k", False, None, [
55             Node("k1", True, 888, None),
56             Node("k2", True, 888, None)]))],
57     Node("c", False, None, [
58         Node("f", True, None, [
59             Node("l", False, None, [
60                 Node("l1", True, 5, None),
61                 Node("l2", True, 888, None)]),
62             Node("m", False, None, [
63                 Node("m1", True, 4, None),
64                 Node("m2", True, 888, None)]))],
65         Node("g", True, None, [
66             Node("n", False, None, [
67                 Node("n1", True, 888, None),
68                 Node("n2", True, 888, None)]),
69             Node("o", False, None, [
70                 Node("o1", True, 888, None),
71                 Node("o2", True, 888, None)]))])])

```

The following is a representation of a **magic-sum game**, where players take turns picking a number in the range $[1, 9]$, and the first player to have 3 numbers that sum to 15 wins. Note that this is a syntactic variant of **tic-tac-toe** or **naughts and crosses**. To see this, consider the numbers on a **magic square** (Figure 14.2); 3 numbers that add to 15 correspond exactly to the winning positions of tic-tac-toe played on the magic square.

masProblem.py — (continued)

```

73
74 class Magic_sum(Node):

```

6	1	8
7	5	3
2	9	4

Figure 14.2: Magic Square

```

75  def __init__(self, xmove=True, last_move=None,
76              available=[1,2,3,4,5,6,7,8,9], x=[], o=[]):
77      """This is a node in the search for the magic-sum game.
78      xmove is True if the next move belongs to X.
79      last_move is the number selected in the last move
80      available is the list of numbers that are available to be chosen
81      x is the list of numbers already chosen by x
82      o is the list of numbers already chosen by o
83      """
84      self.isMax = self.xmove = xmove
85      self.last_move = last_move
86      self.available = available
87      self.x = x
88      self.o = o
89      self.allchildren = None #computed on demand
90      lm = str(last_move)
91      self.name = "start" if not last_move else "o="+lm if xmove else
          "x="+lm
92
93  def children(self):
94      if self.allchildren is None:
95          if self.xmove:
96              self.allchildren = [
97                  Magic_sum(xmove = not self.xmove,
98                          last_move = sel,
99                          available = [e for e in self.available if e is
100                              not sel],
101                          x = self.x+[sel],
102                          o = self.o)
103                  for sel in self.available]
104          else:
105              self.allchildren = [
106                  Magic_sum(xmove = not self.xmove,
107                          last_move = sel,
108                          available = [e for e in self.available if e is
109                              not sel],
110                          x = self.x,
111                          o = self.o+[sel])
112                  for sel in self.available]
113      return self.allchildren
114
115  def isLeaf(self):
116      """A leaf has no numbers available or is a win for one of the

```

```

115         players.
116         We only need to check for a win for o if it is currently x's turn,
117         and only check for a win for x if it is o's turn (otherwise it would
118         have been a win earlier).
119         """
120         return (self.available == [] or
121                 (sum_to_15(self.last_move,self.o)
122                  if self.xmove
123                  else sum_to_15(self.last_move,self.x)))
124
125     def evaluate(self):
126         if self.xmove and sum_to_15(self.last_move,self.o):
127             return -1
128         elif not self.xmove and sum_to_15(self.last_move,self.x):
129             return 1
130         else:
131             return 0
132
133     def sum_to_15(last,selected):
134         """is true if last, together with two other elements of selected sum to
135         15.
136         """
137         return any(last+a+b == 15
138                    for a in selected if a != last
139                    for b in selected if b != last and b != a)

```

14.1.2 Minimax and α - β Pruning

This is a naive depth-first **minimax algorithm** that searches the whole tree:

```

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

```

Testing:

```

_____masMiniMax.py — (continued)_____
70 from masProblem import fig10_5, Magic_sum, Node
71
72 # Node.max_display_level=2 # print detailed trace
73 # minimax_alpha_beta(fig10_5, -9999, 9999,0)
74 # minimax_alpha_beta(Magic_sum(), -9999, 9999,0)

```

```

75 |
76 | #To test much time alpha-beta pruning can save over minimax:
77 | ## import timeit
78 | ## timeit.Timer("minimax(Magic_sum(),0)",setup="from __main__ import
    | minimax, Magic_sum").timeit(number=1)
79 | ## timeit.Timer("minimax_alpha_beta(Magic_sum(), -9999, 9999,0)",
    | setup="from __main__ import minimax_alpha_beta,
    | Magic_sum").timeit(number=1)

```

Exercise 14.1 In the magic-sum game, a state is represented as lists of moves. The same state could be reached by more than one sequence of moves. Change the representation of the game and/or the search procedures to recognize when the value of a state has already been computed. How much does this improve the search?

Exercise 14.2 There are symmetries in tic-tac toe, such as rotation and reflection. How can the representation and/or the algorithm be changed to recognize symmetries? How much difference does it make?

14.2 Multiagent Learning

The next code is for multiple agents that learn when interacting with other agents. The main difference from the simulator of the last chapter is that the games take actions from all the agents and provide a separate reward to each agent. Any of the reinforcement learning agents from the last chapter can be used.

14.2.1 Simulating Multiagent Interaction with an Environment

A game has a name, a list of player roles (which are strings for printing), a list of lists of actions (actions[i][j] is the jth action for agent i), a list of states, and an initial state. The default is to have a single state, and the initial state is a randomly selected state.

```

masLearn.py — Multiagent learning
11 | import random
12 | from display import Displayable
13 | import matplotlib.pyplot as plt
14 | from rlProblem import RL_agent
15 |
16 | class Game(Displayable):
17 |     def __init__(self, name, players, actions, states=['s0'],
    |         initial_state=None):
18 |         self.name = name
19 |         self.players = players # list of roles (strings) of the players
20 |         self.num_players = len(players)
21 |         self.actions = actions # action[i] is list of actions for agent i

```

```

22     self.states = states # list of environment states; default single
        state
23     if initial_state is None:
24         self.initial_state = random.choice(states)
25     else:
26         self.initial_state = initial_state

```

The simulation for a game passes the joint action from all the agents to the environment, which returns a tuple of rewards – one for each agent – and the next state.

```

masLearn.py — (continued)
28 def sim(self, ag_types, discount=0):
29     """returns a simulation using default values for agent types
30     (This is a simple interface to SimulateGame)
31     ag_types is a list of agent functions (one for each player in the
        game)
32     The default is for one-off games where discount=0
33     """
34     return SimulateGame(self,
35                          [ag_types[i](ag_types[i].__name__,
36                                       self.actions[i], discount)
37                           for i in range(self.num_players)])
38 class SimulateGame(Displayable):
39     """A simulation of a game.
40     (This is not subclass of a game, as a game can have multiple games.)
41     """
42     def __init__(self, game, agents):
43         """ Simulates game
44         agents is a list of agents, one for each player in the game
45         """
46         #self.max_display_level = 3
47         self.game = game
48         self.agents = agents
49         # Collect Statistics:
50         self.action_counts = [{act:0 for act in game.actions[i]} for i in
            range(game.num_players)]
51         self.reward_sum = [0 for i in range(game.num_players)]
52         self.dist = {}
53         self.dist_history = []
54         self.actions = tuple(ag.initial_action(game.initial_state) for ag
            in self.agents)
55         self.num_steps = 0
56
57     def go(self, steps):
58         for i in range(steps):
59             self.num_steps += 1
60             (rewards, state) = self.game.play(self.actions)
61             self.display(3, f"In go {rewards=}, {state=}")

```

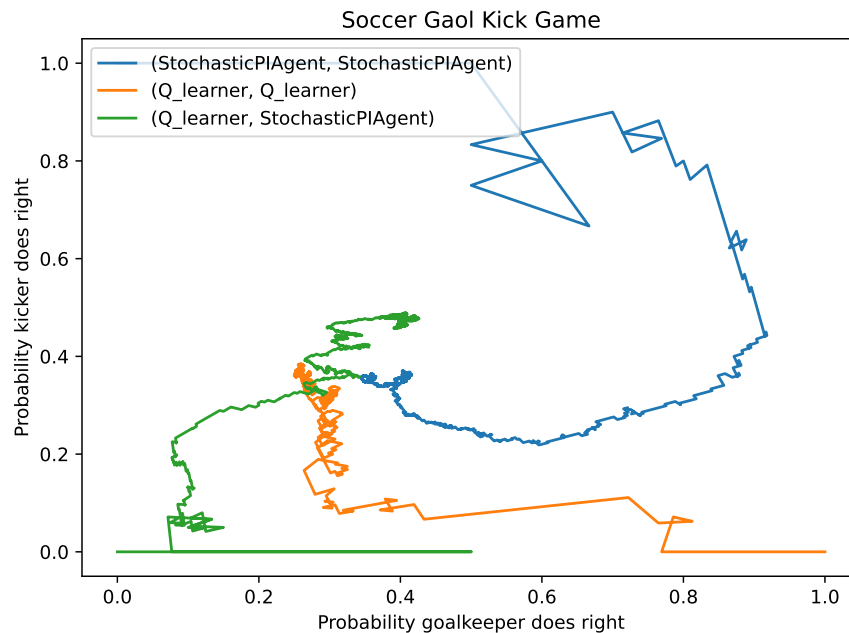



Figure 14.3: Dynamics of three runs of SoccerGame

```

62         self.reward_sum = [self.reward_sum[i]+rewards[i] for i in
63                             range(len(rewards))]
64         self.actions = tuple(agent.select_action(reward, state)
65                               for (agent,reward) in
66                                   zip(self.agents,rewards))
67         for i in range(self.game.num_players):
68             self.action_counts[i][self.actions[i]] += 1
69         self.dist_history.append([a:i/self.num_steps for (a,i) in
70                                 elt.items()])
71         for elt in self.action_counts])
72         self.display(1,"Scores:", ' '.join(
73             f"{self.agents[i].name} average
74             reward={self.reward_sum[i]/self.num_steps}"
75             for i in range(self.game.num_players)))
76         self.display(1,"Distributions:",
77                     ' '.join(str({a:self.dist_history[-1][i][a]
78                                   /sum(self.dist_history[-1][i].values())
79                                   for a in self.game.actions[i]})
80                               for i in range(self.game.num_players)))

```

The plot shows how the empirical distributions of two actions by two agents changes as the learning continues.

Figure 14.3 shows the plot of 3 runs. The first (blue) run, where both agents are running stochastic policy iteration, starts with the goalkeeper going left

and the kicker going right; it ends with both probabilities around 0.35. The second (orange) run, where both agents are doing Q-learning, starts with the goalkeeper going right and the kicker going left; it ends with empirical probabilities of 0.24 for the goalkeeper going right and 0.36 for the kicker going right. The third (green) run, where the goalkeeper is doing Q-learning and the kicker is doing stochastic policy iteration, starts both players going left; it ends with empirical probabilities of 0.41 for the goalkeeper going right and 0.46 for the kicker going right. (You can tell the start as the empirical distribution starts with 0 or 1 probabilities, and moves quickly initially.) This figure is generated using the commented out code at the end of `masLearn.py`.

```

masLearn.py — (continued)
78 def plot_dynamics(self, x_ag=0, y_ag=1, x_action=0, y_action=0):
79     """ plot how the empirical probabilities vary
80     x_ag index of the agent on the x-axis
81     y_ag index of the agent on the y-axis
82     x_action index of the action plotted for x_ag
83     y_action index of the action plotted for y_ag
84     """
85     plt.ion() # make it interactive
86     plt.title(self.game.name)
87     x_act = self.game.actions[x_ag][x_action]
88     y_act = self.game.actions[y_ag][y_action]
89     plt.xlabel(f"Probability {self.game.players[x_ag]} does "
90              f"{self.agents[x_ag].actions[x_action]}")
91     plt.ylabel(f"Probability {self.game.players[y_ag]} does "
92              f"{self.agents[y_ag].actions[y_action]}")
93     plt.plot([self.dist_history[i][x_ag][x_act]
94              for i in range(len(self.dist_history))],
95              [self.dist_history[i][y_ag][y_act]
96              for i in range(len(self.dist_history))],
97              label = f"({self.agents[x_ag].name},
98                   {self.agents[y_ag].name})")
99     plt.legend()
100    plt.show()

```

14.2.2 Example Games

The following are games from Poole and Mackworth [2023].

```

masLearn.py — (continued)
101 class ShoppingGame(Game):
102     def __init__(self):
103         Game.__init__(self, "Shopping Game",
104                       ['football-preferrer', 'shopping-preferrer'], #players
105                       [['shopping', 'football']]*2 # actions
106                       )
107
108     def play(self, actions):

```

```

109         """Given (action1,action2) returns (resulting_state, (reward1,
110             reward2))
111         """
112         return ({('football', 'football'): (2, 1),
113             ('football', 'shopping'): (0, 0),
114             ('shopping', 'football'): (0, 0),
115             ('shopping', 'shopping'): (1, 2)
116             }[actions], 's')
117
118 class SoccerGame(Game):
119     def __init__(self):
120         Game.__init__(self, "Soccer Gaol Kick Game",
121             ['goalkeeper', 'kicker'], # players
122             [['right', 'left']]*2 # actions
123         )
124
125     def play(self, actions):
126         """Given (action1,action2) returns (resulting_state, (reward1,
127             reward2))
128         resulting state is 's'
129         """
130         return ({('left', 'left'): (0.6, 0.4),
131             ('left', 'right'): (0.3, 0.7),
132             ('right', 'left'): (0.2, 0.8),
133             ('right', 'right'): (0.9,0.1)
134             }[actions], 's')
135
136 class GameShow(Game):
137     def __init__(self):
138         Game.__init__(self, "Game Show (prisoners dilemma)",
139             ['Agent 1', 'Agent 2'], # players
140             [['takes', 'gives']]*2 # actions
141         )
142
143     def play(self, actions):
144         return ({('takes', 'takes'): (1, 1),
145             ('takes', 'gives'): (11, 0),
146             ('gives', 'takes'): (0, 11),
147             ('gives', 'gives'): (10, 10)
148             }[actions], 's')
149
150 class UniqueNEGameExample(Game):
151     def __init__(self):
152         Game.__init__(self, "3x3 Unique NE Game Example",
153             ['agent 1', 'agent 2'], # players
154             [['a1', 'b1', 'c1'], ['d2', 'e2', 'f2']]
155         )
156
157     def play(self, actions):
158         return ({('a1', 'd2'): (3, 5),

```

```

157         ('a1', 'e2'): (5, 1),
158         ('a1', 'f2'): (1, 2),
159         ('b1', 'd2'): (1, 1),
160         ('b1', 'e2'): (2, 9),
161         ('b1', 'f2'): (6, 4),
162         ('c1', 'd2'): (2, 6),
163         ('c1', 'e2'): (4, 7),
164         ('c1', 'f2'): (0, 8)
165     }[actions], 's')

```

14.2.3 Testing Games and Environments

```

masLearn.py — (continued)
167 # Choose a game:
168 # gm = ShoppingGame()
169 # gm = SoccerGame()
170 # gm = GameShow()
171 # gm = UniqueNEGameExample()
172
173 from rlQLearner import Q_learner
174 from rlProblem import RL_agent
175 from rlStochasticPolicy import StochasticPIAgent
176 # Choose one of the combinations of learners:
177 # sm = gm.sim([StochasticPIAgent, StochasticPIAgent]); sm.go(10000)
178 # sm = gm.sim([Q_learner, Q_learner]); sm.go(10000)
179 # sm = gm.sim([Q_learner, StochasticPIAgent]); sm.go(10000)
180 # sm = gm.sim([StochasticPIAgent, Q_learner]); sm.go(10000)
181
182 # sm.plot_dynamics()

```

Exercise 14.3 Consider a pair of controllers for a games (try multiple controllers and games, including the soccer game). Does the empirical distribution represent a Nash equilibrium? Would either agent be better off if they played a Nash equilibrium instead of the empirical distribution? [10000 steps might not be enough for the algorithm to converge.]

Exercise 14.4 Try the Game Show (prisoner's dilemma) with two StochasticPIAgent agents and $\alpha_{\text{fun}} = \lambda$ $k: 0.1$, and also with other values of k , including 0.01 . Do different values of k work qualitatively differently? Explain why. Is one better? Try other games and other algorithms.

Exercise 14.5 Consider the alternative ways to implement stochastic policy iteration of Exercise 13.4.

- What value(s) of c converge for the soccer game? Explain your results.
- Suggest another method that works well for the soccer game, the other games and other RL environments.

Exercise 14.6 For the soccer game, how can a Q_learner be regularly beaten? Assume that the random number generator is secret. (Hint: can you predict what

it will do?) What happens when it is played against an adversary that knows how it learns? What happens if two of these agents are played against each other? Can a StochasticPIAgent be defeated in the same way?

Individuals and Relations

Here we implement top-down proofs for Datalog and logic programming. This is much less efficient than Prolog, which is typically implemented by compiling to an abstract machine. If you want to do serious work, we suggest using Prolog; SWI Prolog (<https://www.swi-prolog.org>) is good.

15.1 Representing Datalog and Logic Programs

The following extends the knowledge bases of Chapter 5 to include logical variables. In that chapter, atoms did not have structure and were represented as strings. Here atoms can have arguments including variables (defined below) and constants (represented by strings).

Function symbols have the same representation as atoms. To make unification simpler and to allow treating clauses as data, `Func` is defined as an abbreviation for `Atom`.

```
_____logicRelation.py — Datalog and Logic Programs _____  
11 from display import Displayable  
12 import logicProblem  
13  
14 class Var(Displayable):  
15     """A logical variable"""  
16     def __init__(self, name):  
17         """name"""  
18         self.name = name  
19  
20     def __str__(self):  
21         return self.name  
22     __repr__ = __str__  
23
```

```

24     def __eq__(self, other):
25         return isinstance(other, Var) and self.name == other.name
26     def __hash__(self):
27         return hash(self.name)
28
29     class Atom(object):
30         """An atom"""
31         def __init__(self, name, args):
32             self.name = name
33             self.args = args
34
35         def __str__(self):
36             return f"{self.name}({','.join(str(a) for a in self.args)})"
37         __repr__ = __str__
38
39 Func = Atom # same syntax is used for function symbols

```

The following extends Clause of Section 5.1 to include also a set of logical variables in the clause. It also allows for atoms that are strings (as in Chapter 5) and makes them into atoms.

```

                                     logicRelation.py — (continued)
41 class Clause(logicProblem.Clause):
42     next_index=0
43     def __init__(self, head, *args, **nargs):
44         if not isinstance(head, Atom):
45             head = Atom(head)
46         logicProblem.Clause.__init__(self, head, *args, **nargs)
47         self.logical_variables = log_vars([self.head, self.body], set())
48
49     def rename(self):
50         """create a unique copy of the clause"""
51         if self.logical_variables:
52             sub = {v:Var(f"{v.name}_{Clause.next_index}") for v in
53                  self.logical_variables}
54             Clause.next_index += 1
55             return Clause(apply(self.head, sub), apply(self.body, sub))
56         else:
57             return self
58
59     def log_vars(exp, vs):
60         """the union the logical variables in exp and the set vs"""
61         if isinstance(exp, Var):
62             return {exp}|vs
63         elif isinstance(exp, Atom):
64             return log_vars(exp.name, log_vars(exp.args, vs))
65         elif isinstance(exp, (list, tuple)):
66             for e in exp:
67                 vs = log_vars(e, vs)
68         return vs

```


15.2 Unification

The unification algorithm is very close to the pseudocode of Section 15.5.3 of Poole and Mackworth [2023].

```

logicRelation.py — (continued)
69 unifdisp = Var(None) # for display
70
71 def unify(t1,t2):
72     e = [(t1,t2)]
73     s = {} # empty dictionary
74     while e:
75         (a,b) = e.pop()
76         unifdisp.display(2,f'unifying{(a,b)}, e={e},s={s}')
77         if a != b:
78             if isinstance(a,Var):
79                 e = apply(e,{a:b})
80                 s = apply(s,{a:b})
81                 s[a]=b
82             elif isinstance(b,Var):
83                 e = apply(e,{b:a})
84                 s = apply(s,{b:a})
85                 s[b]=a
86             elif isinstance(a,Atom) and isinstance(b,Atom) and
87                 a.name==b.name and len(a.args)==len(b.args):
88                 e += zip(a.args,b.args)
89             elif isinstance(a,(list,tuple)) and isinstance(b,(list,tuple))
90                 and len(a)==len(b ):
91                 e += zip(a,b)
92             else:
93                 return False
94         return s
95
96 def apply(e,sub):
97     """e is an expression
98     sub is a {var:val} dictionary
99     returns e with all occurrence of var replaces with val"""
100     if isinstance(e,Var) and e in sub:
101         return sub[e]
102     if isinstance(e,Atom):
103         return Atom(e.name, apply(e.args,sub))
104     if isinstance(e,list):
105         return [apply(a,sub) for a in e]
106     if isinstance(e,tuple):
107         return tuple(apply(a,sub) for a in e)
108     if isinstance(e,dict):
109         return {k:apply(v,sub) for (k,v) in e.items()}
110     else:
111         return e

```

Test cases:

```

111 ##### Test cases:
112 # unidisp.max_display_level = 2 # show trace
113 e1 = Atom('p',[Var('X'),Var('Y'),Var('Y')])
114 e2 = Atom('p',['a',Var('Z'),'b'])
115 # apply(e1,{Var('Y'):'b'})
116 # unify(e1,e2)
117 e3 = Atom('p',['a',Var('Y'),Var('Y')])
118 e4 = Atom('p',[Var('Z'),Var('Z'),'b'])
119 # unify(e3,e4)

```

15.3 Knowledge Bases

The following modifies KB of Section 5.1 so that clause indexing is only on the predicate symbol of the head of clauses.

```

121 class KB(logicProblem.KB):
122     """A first-order knowledge base.
123     only the indexing is changed to index on name of the head."""
124
125     def add_clause(self, c):
126         """Add clause c to clause dictionary"""
127         if c.head.name in self.atom_to_clauses:
128             self.atom_to_clauses[c.head.name].append(c)
129         else:
130             self.atom_to_clauses[c.head.name] = [c]

```

simp_KB is the simple knowledge base of Figure 15.1 of Poole and Mackworth [2023].

```

11 from logicRelation import Var, Atom, Clause, KB
12
13 simp_KB = KB([
14     Clause(Atom('in',['kim','r123'])),
15     Clause(Atom('part_of',['r123','cs_building'])),
16     Clause(Atom('in',[Var('X'),Var('Y')]),
17             [Atom('part_of',[Var('Z'),Var('Y')]),
18              Atom('in',[Var('X'),Var('Z')])])
19 ])

```

elect_KB is the relational version of the knowledge base for the electrical system of a house, as described in Example 15.11 of Poole and Mackworth [2023].

```

21 # define abbreviations to make the clauses more readable:
22 def lit(x): return Atom('lit',[x])
23 def light(x): return Atom('light',[x])

```

```

24 def ok(x): return Atom('ok',[x])
25 def live(x): return Atom('live',[x])
26 def connected_to(x,y): return Atom('connected_to',[x,y])
27 def up(x): return Atom('up',[x])
28 def down(x): return Atom('down',[x])
29
30 L = Var('L')
31 W = Var('W')
32 W1 = Var('W1')
33
34 elect_KB = KB([
35     # lit(L) is true if light L is lit.
36     Clause(lit(L),
37             [light(L),
38              ok(L),
39              live(L)]),
40
41     # live(W) is true if W is live (i.e., current will flow through it)
42     Clause(live(W),
43             [connected_to(W,W1),
44              live(W1)]),
45
46     Clause(live('outside')),
47
48     # light(L) is true if L is a light
49     Clause(light('l1')),
50     Clause(light('l2')),
51
52     # connected_to(W0,W1) is true if W0 is connected to W1 such that
53     # current will flow from W1 to W0.
54
55     Clause(connected_to('l1','w0')),
56     Clause(connected_to('w0','w1'),
57             [ up('s2'), ok('s2')]),
58     Clause(connected_to('w0','w2'),
59             [ down('s2'), ok('s2')]),
60     Clause(connected_to('w1','w3'),
61             [ up('s1'), ok('s1')]),
62     Clause(connected_to('w2','w3'),
63             [ down('s1'), ok('s1')]),
64     Clause(connected_to('l2','w4')),
65     Clause(connected_to('w4','w3'),
66             [ up('s3'), ok('s3')]),
67     Clause(connected_to('p1','w3')),
68     Clause(connected_to('w3','w5'),
69             [ ok('cb1')]),
70     Clause(connected_to('p2','w6')),
71     Clause(connected_to('w6','w5'),
72             [ ok('cb2')]),
73     Clause(connected_to('w5','outside'),

```

```

74         [ ok('outside_connection')]],
75
76     # up(S) is true if switch S is up
77     # down(S) is true if switch S is down
78     Clause(down('s1')),
79     Clause(up('s2')),
80     Clause(up('s3')),
81
82     # ok(L) is true if K is working. Everything is ok:
83     Clause(ok(L)),
84 ])
```

15.4 Top-down Proof Procedure

The top-down proof procedure is the one defined in Section 15.5.4 of Poole and Mackworth [2023] and shown in Figure 15.5. It is like `prove` defined in Section 5.3. It implements the iterator interface so that answers can be generated one at a time (or put in a list), and returns answers. To implement “choose” it loops over all alternatives and *yields* (returns one element at a time) the successful proofs.

```

_____logicRelation.py — (continued)_____
132 def ask(self, query):
133     """self is the current KB
134     query is a list of atoms to be proved
135     generates {variable:value} dictionary"""
136
137     qvars = list(log_vars(query, set()))
138     for ans in self.prove(qvars, query):
139         yield {x:v for (x,v) in zip(qvars,ans)}
140
141 def ask_all(self, query):
142     """returns a list of all answers to the query given kb"""
143     return list(self.ask(query))
144
145 def ask_one(self, query):
146     """returns an answer to the query given kb or None of there are no
147     answers"""
148     for ans in self.ask(query):
149         return ans
150
151 def prove(self, ans, ans_body, indent=""):
152     """enumerates the proofs for ans_body
153     ans_body is a list of atoms to be proved
154     ans is the list of values of the query variables
155     """
156     self.display(2,indent,f"(yes({ans}) <-"," & ".join(str(a) for a in
157         ans_body))
```

```

156         if ans_body==[]:
157             yield ans
158         else:
159             selected, remaining = self.select_atom(ans_body)
160             if self.built_in(selected):
161                 yield from self.eval_built_in(ans, selected, remaining,
162                                         indent)
163             else:
164                 for chosen_clause in self.atom_to_clauses[selected.name]:
165                     clause = chosen_clause.rename() # rename variables
166                     sub = unify(selected, clause.head)
167                     if sub is not False:
168                         self.display(3,indent,"KB.prove: selected=",
169                                     selected, "clause=",clause,"sub=",sub)
170                         resans = apply(ans,sub)
171                         new_ans_body = apply(clause.body+remaining, sub)
172                         yield from self.prove(resans, new_ans_body, indent+"
173                                     ")
174
175     def select_atom(self,lst):
176         """given list of atoms, return (selected atom, remaining atoms)
177         """
178         return lst[0],lst[1:]
179
180     def built_in(self,atom):
181         return atom.name in ['lt','triple']
182
183     def eval_built_in(self,ans, selected, remaining, indent):
184         if selected.name == 'lt': # less than
185             [a1,a2] = selected.args
186             if a1 < a2:
187                 yield from self.prove(ans, remaining, indent+" ")
188         if selected.name == 'triple': # use triple store (AIFCA Ch 16)
189             yield from self.eval_triple(ans, selected, remaining, indent)

```

The unit test run when loading is the query $in(A,B)$, from `simp_KB`. It should have two answers.

```

relnExamples.py — (continued)
86 # Example Queries:
87 # simp_KB.max_display_level = 2 # show trace
88 # ask_all(simp_KB, [Atom('in',[Var('A'),Var('B')])])
89
90 A = Var('A')
91 B = Var('B')
92
93 def test_ask_all(kb=simp_KB,
94                 query=[Atom('in',[A,B])],
95                 res=[{ A:'kim',B:'r123'}, {A:'kim',B: 'cs_building'}]):
96     ans= kb.ask_all(query)
97     assert ans == res, f"ask_all({query}) gave answer {ans}"

```

```

98     print("ask_all: Passed unit test")
99
100 if __name__ == "__main__":
101     test_ask_all()
102
103 # elect_KB.max_display_level = 2 # show trace
104 # elect_KB.ask_all([light('l1')])
105 # elect_KB.ask_all([light('l6')])
106 # elect_KB.ask_all([up(Var('X'))])
107 # elect_KB.ask_all([connected_to('w0',W)])
108 # elect_KB.ask_all([connected_to('w1',W)])
109 # elect_KB.ask_all([connected_to(W,'w3')])
110 # elect_KB.ask_all([connected_to(W1,W)])
111 # elect_KB.ask_all([live('w6')])
112 # elect_KB.ask_all([live('p1')])
113 # elect_KB.ask_all([Atom('lit',[L])])
114 # elect_KB.ask_all([Atom('lit',['l2']), live('p1')])
115 # elect_KB.ask_all([live(L)])

```

Exercise 15.1 Implement ask-the-user similar to Section 5.3. Augment this by allowing the user to specify which instances satisfy an atom. For example, by asking the user “for what X is w1 connected to X?”; or perhaps in a more user friendly way.

15.5 Logic Program Example

The following is an append program and the query of Example 15.30 of Poole and Mackworth [2023].

```

append(nil,W,W).
append(c(A,X),Y,c(A,Z)) <-
    append(X,Y,Z).

```

The term $c(A,X)$ is represented using Atom
In Prolog syntax:

```

append(nil,W,W).
append([A|X],Y,[A|Z]) :-
    append(X,Y,Z).

```

The value if `lst` is `[l,i,s,t]`. The query is

```
? append(F,[L],[l,i,s,t]).
```

We first define some constants and functions to make it more readable.

```

_____logicRelation.py — (continued)_____
188 A = Var('A')
189 F = Var('F')
190 L = Var('L')
191 W = Var('W')

```

```
192 X = Var('X')
193 Y = Var('Y')
194 Z = Var('Z')
195 def cons(h,t): return Atom('cons',[h,t])
196 def append(a,b,c): return Atom('append',[a,b,c])
197
198 app_KB = KB([
199     Clause(append('nil',W,W)),
200     Clause(append(cons(A,X), Y,cons(A,Z)),
201             [append(X,Y,Z)])
202 ])
203
204
205 lst = cons('l',cons('i',cons('s',cons('t','nil'))))
206 # app_KB.max_display_level = 2 #show derivation
207 #app_KB.ask_all([append(F,cons(A,'nil'), lst)])
208 # Think about the expected answer before trying:
209 #app_KB.ask_all([append(X, Y, lst)])
210 #app_KB.ask_all([append(lst, lst, L), append(X, cons('s',Y), L)])
```


Knowledge Graphs and Ontologies

16.1 Triple Store

A triple store provides efficient indexing for triples. For any combination of the subject-verb-object being provided or not, it can efficiently retrieve the corresponding triples. This should be comparable in speed to commercial triple stores, but would probably handle fewer triples, as it is not optimized for space. It also have fewer bells and whistles (e.g., ways to visualize triples and traverse the graph).

A triple store implements an index that covers all cases of where the subject, verb, or object are provided or not. The unspecified parts are given using Q (with value '?'). Thus, for example, `index[(Q, vrb, Q)]` is the list of triples with verb `vrb`. `index[(sub, Q, obj)]` is the list of triples with subject `sub` and object `obj`.

```
knowledgeGraph.py — Knowledge graph triple store
11 from display import Displayable
12
13 class TripleStore(Displayable):
14     Q = '?' # query position
15
16     def __init__(self):
17         self.index = {}
18
19     def add(self, triple):
20         (sb,vb,ob) = triple
21         Q = self.Q # make it easier to read
22         add_to_index(self.index, (Q,Q,Q), triple)
```

```

23     add_to_index(self.index, (Q,Q,ob), triple)
24     add_to_index(self.index, (Q,vb,Q), triple)
25     add_to_index(self.index, (Q,vb,ob), triple)
26     add_to_index(self.index, (sb,Q,Q), triple)
27     add_to_index(self.index, (sb,Q,ob), triple)
28     add_to_index(self.index, (sb,vb,Q), triple)
29     add_to_index(self.index, triple, triple)
30
31     def __len__(self):
32         """number of triples in the triple store"""
33         return len(self.index[(Q,Q,Q)])

```

The lookup method returns a list of triples that match a pattern. The pattern is a triple of the form (i, j, k) where each of i , j , and k is either “Q” or a given value; specifying whether the subject, verb, and object are provided in the query or not. `lookup((Q,Q,Q))` returns all triples. `lookup((s,v,o))` can be used to check whether the triple (s, v, o) is in the triple store; it returns `[]` if the triple is not in the knowledge graph, and `[(s, v, o)]` if it is.

```

_____knowledgeGraph.py — (continued)_____
35     def lookup(self, query):
36         """pattern is a triple of the form (i,j,k) where
37             each i, j, k is either Q or a value for the
38             subject, verb and object respectively.
39             returns all triples with the specified non-Q vars in corresponding
               position
40         """
41         if query in self.index:
42             return self.index[query]
43         else:
44             return []
45
46     def add_to_index(dict, key, value):
47         if key in dict:
48             dict[key].append(value)
49         else:
50             dict[key] = [value]

```

Here is a simple test triple store. In Wikidata Q262802 denotes the football (soccer) player Christine Sinclair, P27 is the country of citizenship, and Q16 is Canada.

```

_____knowledgeGraph.py — (continued)_____
52 # test cases:
53 sts = TripleStore() # simple triple store
54 Q = TripleStore.Q # makes it easier to read
55 sts.add(('entity/Q262802', 'http://schema.org/name', "Christine Sinclair"))
56 sts.add(('entity/Q262802', '/prop/direct/P27', 'entity/Q16'))
57 sts.add(('entity/Q16', 'http://schema.org/name', "Canada"))
58
59 # sts.lookup(('entity/Q262802', Q, Q))

```

```

60 # sts.lookup((Q, 'http://schema.org/name', Q))
61 # sts.lookup((Q, 'http://schema.org/name', "Canada"))
62 # sts.lookup(('entity/Q16', 'http://schema.org/name', "Canada"))
63 # sts.lookup(('entity/Q262802', 'http://schema.org/name', "Canada"))
64 # sts.lookup((Q, Q, Q))
65
66 def test_kg(kg=sts, q=('entity/Q262802', Q, Q),
        res=[('entity/Q262802', 'http://schema.org/name', "Christine
        Sinclair"), ('entity/Q262802', '/prop/direct/P27', 'entity/Q16')]):
67     """Knowledge graph unit test"""
68     ans = kg.lookup(q)
69     assert res==ans, f"test_kg answer {ans}"
70     print("knowledge graph unit test passed")
71
72 if __name__ == "__main__":
73     test_kg()

```

To read rdf files, you can use rdflib (<https://rdflib.readthedocs.io/en/stable/>).

The default in `load_file` is to include only English names; multiple languages can be included in the list. If the language restriction is `None`, all tuples are included. Converting to strings, as done here, loses information, e.g., the language associated with the literals. If you don't want to lose information, you can use rdflib objects, by omitting `str` in the call to `ts.add`.

knowledgeGraph.py — (continued)

```

75 # before using do:
76 # pip install rdflib
77
78 def load_file(ts, filename, language_restriction=['en']):
79     import rdflib
80     g = rdflib.Graph()
81     g.parse(filename)
82     for (s,v,o) in g:
83         if language_restriction and isinstance(o, rdflib.term.Literal) and
            o._language and o._language not in language_restriction:
84             pass
85         else:
86             ts.add((str(s), str(v), str(o)))
87     print(f"{len(g)} triples read. Triple store has {len(ts)} triples.")
88
89 TripleStore.load_file = load_file
90
91 ##### Test cases #####
92 ts = TripleStore()
93 #ts.load_file('http://www.wikidata.org/wiki/Special:EntityData/Q262802.nt')
94 q262802 = 'http://www.wikidata.org/entity/Q262802'
95 #res=ts.lookup((q262802, 'http://www.wikidata.org/prop/P27', Q)) # country
            of citizenship
96 # The attributes of the object in the first answer to the above query:

```

```

97 | #ts.lookup((res[0][2],Q,Q))
98 | #ts.lookup((q262802, 'http://www.wikidata.org/prop/P54',Q)) # member of
    | sports team
99 | #ts.lookup((q262802, 'http://schema.org/name',Q))

```

16.2 Integrating Datalog and Triple Store

The following extends the definite clause reasoner in the previous chapter to include a built-in “triple” predicate (an atom with name “triple” and three arguments). The instances of this predicate are retrieved from the triple store. This is a simplified version of what can be done with the `semweb` library of SWI Prolog ([https://www.swi-prolog.org/pldoc/doc_for?object=section\(%27packages/semweb.html%27\)](https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/semweb.html%27))). For anything serious, we suggest you use that. Note that the `semweb` library uses “`rdf`” as the predicate name, and Poole and Mackworth [2023] uses “`prop`” in Section 16.1.3 for the same predicate as “triple”.

```

_____knowledgeReasoning.py — Integrating Datalog and triple store _____
11 | from logicRelation import Var, Atom, Clause, KB, unify, apply
12 | from knowledgeGraph import TripleStore, sts
13 | import random
14 |
15 | class KBT(KB):
16 |     def __init__(self, triplestore, statements=[]):
17 |         self.triplestore = triplestore
18 |         KB.__init__(self, statements)
19 |
20 |     def eval_triple(self, ans, selected, remaining, indent):
21 |         query = selected.args
22 |         Q = self.triplestore.Q
23 |         pattern = tuple(Q if isinstance(e,Var) else e for e in query)
24 |         retrieved = self.triplestore.lookup(pattern)
25 |         self.display(3,indent,"eval_triple:
        |         query=",query,"pattern=",pattern,"retrieved=",retrieved)
26 |         for tr in random.sample(retrieved,len(retrieved)):
27 |             sub = unify(tr, query)
28 |             self.display(3,indent,"KB.prove:
        |             selected=",selected,"triple=",tr,"sub=",sub)
29 |             if sub is not False:
30 |                 yield from self.prove(apply(ans,sub), apply(remaining,sub),
        |                 indent+" ")
31 |
32 | # simple test case:
33 | kbt = KBT(sts) # sts is simple triplestore from knowledgeGraph.py
34 | # kbt.ask_all([Atom('triple'),('http://www.wikidata.org/entity/Q262802',
    | Var('P'),Var('O'))]))

```

The following are some larger examples from Wikidata. You must run `load_file` to load the triples related to Christine Sinclair (Q262802). Otherwise the queries won’t work.

The first query is how Christine Sinclair (Q262802) is related to Portland Thorns (Q1446672) with two hops in the knowledge graph. It is asking for a P , O and $P1$ such that

$$(Q262802, P, O) \& (O, P1, Q1446672)$$

```

knowledgeReasoning.py — (continued)
36 O = Var('O'); O1 = Var('O1')
37 P = Var('P')
38 P1 = Var('P1')
39 T = Var('T')
40 N = Var('N')
41 def triple(s,v,o): return Atom('triple',[s,v,o])
42 def lt(a,b): return Atom('lt',[a,b])
43
44 ts = TripleStore()
45 kbts = KBT(ts)
46 #ts.load_file('http://www.wikidata.org/wiki/Special:EntityData/Q262802.nt')
47 q262802 = 'http://www.wikidata.org/entity/Q262802'
48 # How is Christine Sinclair (Q262802) related to Portland Thorns
   (Q1446672) with 2 hops:
49 # kbts.ask_all([triple(q262802, P, O), triple(O, P1,
   'http://www.wikidata.org/entity/Q1446672') ])
```

The second is asking for the name of a team that Christine Sinclair (Q262802) played for. It is asking for a O , T and N , where O is the reified object that gives the relationship, T is the team and N is the name of the team. Informally (with variables staring with uppercase and constants in lower case) this is

$$(q262802, p54, O) \& (O, p54, T) \& (T, name, N)$$

Notice how the reified relation 'P54' (member of sports team) is represented:

```

knowledgeReasoning.py — (continued)
51 # What is the name of a team that Christine Sinclair played for:
52 # kbts.ask_one([triple(q262802, 'http://www.wikidata.org/prop/P54',O),
   triple(O, 'http://www.wikidata.org/prop/statement/P54',T),
   triple(T, 'http://schema.org/name',N)])
```

The third asks for the name of a team that Christine Sinclair (Q262802) played for at two different start times. It is asking for a N , $D1$ and $D2$, N is the name of the team and $D1$ and $D2$ are the start dates. In Wikidata, P54 is “member of sports team” and P580 is “start time”.

```

knowledgeReasoning.py — (continued)
54 # The name of a team that Christine Sinclair played for at two different
   times, and the dates
55 def playedtwice(s,n,d0,d1): return Atom('playedtwice',[s,n,d0,d1])
56 S = Var('S')
57 N = Var('N')
```

```
58 D0 = Var('D0')
59 D1 = Var('D2')
60
61 kbts.add_clause(Clause(playedtwice(S,N,D0,D1), [
62     triple(S, 'http://www.wikidata.org/prop/P54', 0),
63     triple(0, 'http://www.wikidata.org/prop/statement/P54', T),
64     triple(S, 'http://www.wikidata.org/prop/P54', 01),
65     triple(01, 'http://www.wikidata.org/prop/statement/P54', T),
66     lt(0,01), # ensure different and only generated once
67     triple(T, 'http://schema.org/name', N),
68     triple(0, 'http://www.wikidata.org/prop/qualifier/P580', D0),
69     triple(01, 'http://www.wikidata.org/prop/qualifier/P580', D1)
70 ]))
71
72 # kbts.ask_all([playedtwice(q262802,N,D0,D1)])
```

Relational Learning

17.1 Collaborative Filtering

The code here is based on the gradient descent algorithm for matrix factorization of Koren, Bell, and Volinsky [2009].

A rating set consists of training and test data, each a list of *(user, item, rating)* tuples.

```
_____relnCollFilt.py — Latent Property-based Collaborative Filtering_____
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 Rating_set(Displayable):
18     """A rating contains:
19     training_data: list of (user, item, rating) triples
20     test_data: list of (user, item, rating) triples
21     """
22     def __init__(self, training_data, test_data):
23         self.training_data = training_data
24         self.test_data = test_data
```

The following is a representation of Examples 17.5-17.7 of Poole and Mackworth [2023]. This is a much smaller dataset than one would expect to work well.

```
_____relnCollFilt.py — (continued) _____
26 grades_rs = Rating_set( # 3='A', 2='B', 1='C'
27     [('s1','c1',3), # training data
28     ('s2','c1',1),
```

```

29     ('s1','c2',2),
30     ('s2','c3',2),
31     ('s3','c2',2),
32     ('s4','c3',2)],
33 [(['s3','c4',3), # test data
34   ('s4','c4',1)]]

```

A CF_learner does stochastic gradient descent to make a predictor of ratings for user-item pairs.

```

relnCollFilt.py — (continued)
36 class CF_learner(Learner):
37     def __init__(self,
38                   rating_set,          # a Rating_set
39                   step_size = 0.01,    # gradient descent step size
40                   regularization = 1.0, # L2 regularization for full dataset
41                   num_properties = 10, # number of hidden properties
42                   property_range = 0.02 # properties are initialized to be
                                     between
43                                     # -property_range and property_range
44                   ):
45         self.rating_set = rating_set
46         self.training_data = rating_set.training_data
47         self.test_data = self.rating_set.test_data
48         self.step_size = step_size
49         self.regularization = regularization
50         self.num_properties = num_properties
51         self.num_ratings = len(self.training_data)
52         self.ave_rating = (sum(r for (u,i,r) in self.training_data)
53                           /self.num_ratings)
54         self.users = {u for (u,i,r) in self.training_data}
55         self.items = {i for (u,i,r) in self.training_data}
56         self.user_bias = {u:0 for u in self.users}
57         self.item_bias = {i:0 for i in self.items}
58         self.user_prop = {u:[random.uniform(-property_range,property_range)
59                               for p in range(num_properties)]
60                           for u in self.users}
61         self.item_prop = {i:[random.uniform(-property_range,property_range)
62                               for p in range(num_properties)]
63                           for i in self.items}
64         # the _delta variables are the changes internal to a batch:
65         self.user_bias_delta = {u:0 for u in self.users}
66         self.item_bias_delta = {i:0 for i in self.items}
67         self.user_prop_delta = {u:[0 for p in range(num_properties)]
68                                 for u in self.users}
69         self.item_prop_delta = {i:[0 for p in range(num_properties)]
70                                 for i in self.items}
71         # zeros is used for users and items not in the training set
72         self.zeros = [0 for p in range(num_properties)]
73         self.epoch = 0
74         self.display(1, "Predict mean:" "(Ave Abs,AveSumSq)",

```



```

75         "training =",self.eval2string(self.training_data,
76         useMean=True),
        "test =",self.eval2string(self.test_data, useMean=True))

```

prediction returns the current prediction of a user on an item.

```

relnCollFilt.py — (continued)
78 def prediction(self,user,item):
79     """Returns prediction for this user on this item.
80     The use of .get() is to handle users or items in test set but not
81     in the training set.
82     """
83     if user in self.user_bias: # user in training set
84         if item in self.item_bias: # item in training set
85             return (self.ave_rating
86                     + self.user_bias[user]
87                     + self.item_bias[item]
88                     + sum([self.user_prop[user][p]*self.item_prop[item][p]
89                           for p in range(self.num_properties)]))
89         else: # training set contains user but not item
90             return (self.ave_rating + self.user_bias[user])
91     elif item in self.item_bias: # training set contains item but not
92         user
93         return self.ave_rating + self.item_bias[item]
94     else:
95         return self.ave_rating

```

learn carries out num_epochs epochs of stochastic gradient descent with batch_size giving the number of training examples in a batch. The number of epochs is approximately the average number of times each training data point is used. It is approximate because it processes the integral number of the batch size.

```

relnCollFilt.py — (continued)
96 def learn(self, num_epochs = 50, batch_size=1000):
97     """ do (approximately) num_epochs iterations through the dataset
98     batch_size is the size of each batch of stochastic gradient
99     gradient descent.
100     """
101     batch_size = min(batch_size, len(self.training_data))
102     batch_per_epoch = len(self.training_data) // batch_size #
103     approximate
104     num_iter = batch_per_epoch*num_epochs
105     reglz =
106         self.step_size*self.regularization*batch_size/len(self.training_data)
107         #regularization per batch
108     for i in range(num_iter):
109         if i % batch_per_epoch == 0:
110             self.epoch += 1
111             self.display(1,"Epoch", self.epoch, "(Ave Abs,AveSumSq)",

```

```

109         "training =", self.eval2string(self.training_data),
110         "test =", self.eval2string(self.test_data))
111     # determine errors for a batch
112     for (user,item,rating) in random.sample(self.training_data,
113         batch_size):
114         error = self.prediction(user,item) - rating
115         self.user_bias_delta[user] += error
116         self.item_bias_delta[item] += error
117         for p in range(self.num_properties):
118             self.user_prop_delta[user][p] +=
119                 error*self.item_prop[item][p]
120             self.item_prop_delta[item][p] +=
121                 error*self.user_prop[user][p]
122     # Update all parameters
123     for user in self.users:
124         self.user_bias[user] -=
125             (self.step_size*self.user_bias_delta[user]
126              + reglz*self.user_bias[user])
127         self.user_bias_delta[user] = 0
128         for p in range(self.num_properties):
129             self.user_prop[user][p] -=
130                 (self.step_size*self.user_prop_delta[user][p]
131                  + reglz*self.user_prop[user][p])
132             self.user_prop_delta[user][p] = 0
133     for item in self.items:
134         self.item_bias[item] -=
135             (self.step_size*self.item_bias_delta[item]
136              + reglz*self.item_bias[item])
137         self.item_bias_delta[item] = 0
138         for p in range(self.num_properties):
139             self.item_prop[item][p] -=
140                 (self.step_size*self.item_prop_delta[item][p]
141                  + reglz*self.item_prop[item][p])
142             self.item_prop_delta[item][p] = 0

```

The evaluate method evaluates current predictions on the rating set:

```

relnCollFilt.py — (continued)
137 def evaluate(self, ratings, useMean=False):
138     """returns (average_absolute_error, average_sum_squares_error) for
139         ratings
140     """
141     abs_error = 0
142     sumsq_error = 0
143     if not ratings: return (0,0)
144     for (user,item,rating) in ratings:
145         prediction = self.ave_rating if useMean else
146             self.prediction(user,item)
147         error = prediction - rating
148         abs_error += abs(error)
149         sumsq_error += error * error

```

```

148         return abs_error/len(ratings), sumsq_error/len(ratings)
149
150     def eval2string(self, *args, **nargs):
151         """returns a string form of evaluate, with fewer digits
152         """
153         (abs,ssq) = self.evaluate(*args, **nargs)
154         return f"({abs:.4f}, {ssq:.4f})"

```

Let's test the code on the grades rating set:

```

_____relnCollFilt.py — (continued) _____
156 #lg = CF_learner(grades_rs,step_size = 0.1, regularization = 0.01,
      num_properties = 1)
157 #lg.learn(num_epochs = 500)
158 # lg.item_bias
159 # lg.user_bias
160 # lg.plot_property(0,plot_all=True) # can you explain why?

```

Exercise 17.1 In using `CF_learner` with `grades_rs`, does it work better with 0 properties? Is it overfitting to the data? How can overfitting be adjusted?

Exercise 17.2 Modify the code so that `self.ave_rating` is also learned. It should start as the average rating. Should it be regularized? Does it change from the initialized value? Does it work better or worse?

Exercise 17.3 With the Movielens 100K dataset and the batch size being the whole training set, what happens to the error? How can this be fixed?

Exercise 17.4 Can the regularization avoid iterating through the parameters for all users and items after a batch? Consider items that are in many batches versus those in a few or even no batches. (Warning: This is challenging to get right.)

17.1.1 Plotting

The `plot_predictions` method plots the cumulative distributions for each ground truth. Figure 17.1 shows a plot for the Movielens 100K dataset. Consider the `rating = 1` line. The value for x is the proportion of the predictions with predicted value $\leq x$ when the ground truth has a rating of 1. Similarly for the other lines.

Figure 17.1 is for one run on the training data. What would you expected the test data to look like?

```

_____relnCollFilt.py — (continued) _____
162     def plot_predictions(self, examples="test"):
163         """
164         examples is either "test" or "training" or the actual examples
165         """
166         if examples == "test":
167             examples = self.test_data
168         elif examples == "training":
169             examples = self.training_data

```

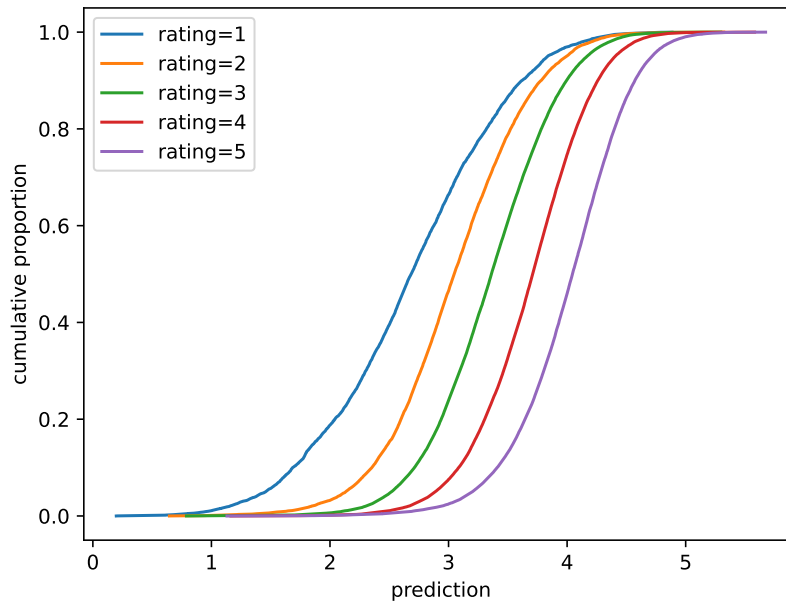


Figure 17.1: learner1.plot_predictions(examples = "training")

```

170 plt.ion()
171 plt.xlabel("prediction")
172 plt.ylabel("cumulative proportion")
173 self.actuals = [[] for r in range(0,6)]
174 for (user,item,rating) in examples:
175     self.actuals[rating].append(self.prediction(user,item))
176 for rating in range(1,6):
177     self.actuals[rating].sort()
178     numrat=len(self.actuals[rating])
179     yvals = [i/numrat for i in range(numrat)]
180     plt.plot(self.actuals[rating], yvals,
181             label="rating="+str(rating))
181 plt.legend()
182 plt.draw()

```

The `plot_property` method plots a single latent property; see Figure 17.2. Each $(user, item, rating)$ is plotted where the x -value is the value of the property for the user, the y -value is the value of the property for the item, and the rating is plotted at this (x, y) position. That is, $rating$ is plotted at the (x, y) position $(p(user), p(item))$.

Because there are too many ratings to show, `plot_property` selects a random number of points. It is difficult to see what is going on; the `create_top_subset` method was created to show the most rated items and the users who rated the most of these. This should help visualize how the latent property helps.

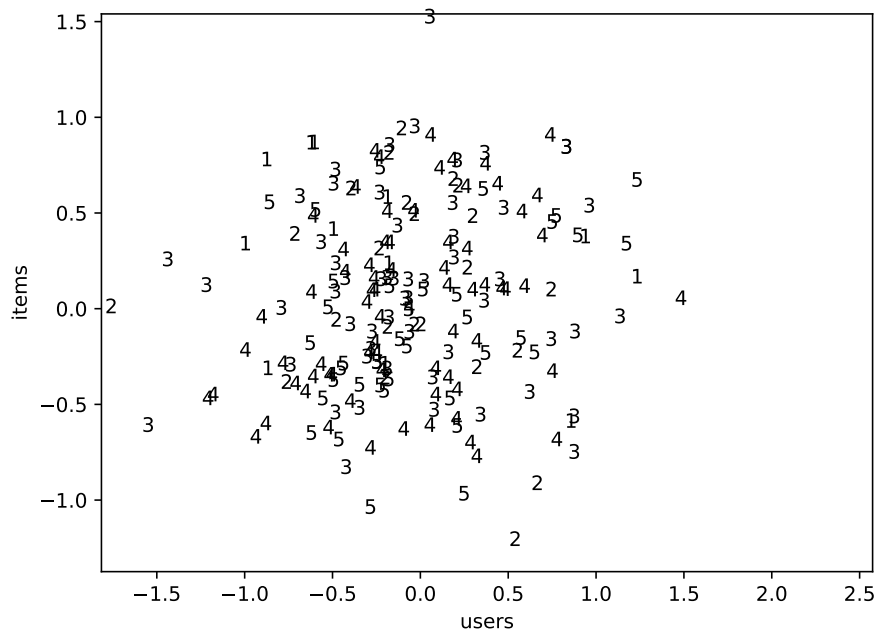


Figure 17.2: learner1.plot_property(0) with 200 random ratings plotted. Rating (u, i, r) has r plotted a position $(p(u), p(i))$ where p is the selected latent property.

```

relnCollFilt.py — (continued)
184 def plot_property(self,
185     p, # property
186     plot_all=False, # true if all points should be plotted
187     num_points=200 # number of random points plotted if not
188     all
189 ):
190     """plot some of the user-movie ratings,
191     if plot_all is true
192     num_points is the number of points selected at random plotted.
193
194     the plot has the users on the x-axis sorted by their value on
195     property p and
196     with the items on the y-axis sorted by their value on property p and
197     the ratings plotted at the corresponding x-y position.
198     """
199     plt.ion()
200     plt.xlabel("users")
201     plt.ylabel("items")
202     user_vals = [self.user_prop[u][p]
203                 for u in self.users]
204     item_vals = [self.item_prop[i][p]

```

```

203         for i in self.items]
204     plt.axis([min(user_vals)-0.02,
205             max(user_vals)+0.05,
206             min(item_vals)-0.02,
207             max(item_vals)+0.05])
208     if plot_all:
209         for (u,i,r) in self.training_data:
210             plt.text(self.user_prop[u][p],
211                     self.item_prop[i][p],
212                     str(r))
213     else:
214         for i in range(num_points):
215             (u,i,r) = random.choice(self.training_data)
216             plt.text(self.user_prop[u][p],
217                     self.item_prop[i][p],
218                     str(r))
219     plt.show()

```

17.1.2 Loading Rating Sets from Files and Websites

This assumes the form of the Movielens datasets Harper and Konstan [2015], available from <http://grouplens.org/datasets/movielens/>.

The Movielens datasets consist of *(user, movie, rating, timestamp)* tuples. The aim here is to predict the future from the past. Tuples with a timestamp before `data_split` form the training set, and those with a timestamp after form the test set.

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

```

relnCollFilt.py — (continued)
221 class Rating_set_from_file(Rating_set):
222     def __init__(self,
223                 date_split=892000000,
224                 local_file=False,
225                 url="http://files.grouplens.org/datasets/movielens/ml-100k/u.data",
226                 file_name="u.data"):
227         self.display(1,"Collaborative Filtering Dataset. Reading...")
228         if local_file:
229             lines = open(file_name,'r')
230         else:
231             lines = (line.decode('utf-8') for line in
232                     urllib.request.urlopen(url))
233         all_ratings = (tuple(int(e) for e in line.strip().split('\t'))
234                        for line in lines)
235         self.training_data = []
236         self.training_stats = {1:0, 2:0, 3:0, 4:0, 5:0}
237         self.test_data = []

```

```

237     self.test_stats = {1:0, 2:0, 3:0, 4:0, 5:0}
238     for (user,item,rating,timestamp) in all_ratings:
239         if timestamp < date_split: # rate[3] is timestamp
240             self.training_data.append((user,item,rating))
241             self.training_stats[rating] += 1
242         else:
243             self.test_data.append((user,item,rating))
244             self.test_stats[rating] += 1
245     self.display(1,"...read:", len(self.training_data),"training
        ratings and",
246                 len(self.test_data),"test ratings")
247     tr_users = {user for (user,item,rating) in self.training_data}
248     test_users = {user for (user,item,rating) in self.test_data}
249     self.display(1,"users:",len(tr_users),"training,",len(test_users),"test,",
250                 len(tr_users & test_users),"in common")
251     tr_items = {item for (user,item,rating) in self.training_data}
252     test_items = {item for (user,item,rating) in self.test_data}
253     self.display(1,"items:",len(tr_items),"training,",len(test_items),"test,",
254                 len(tr_items & test_items),"in common")
255     self.display(1,"Rating statistics for training set:
        ",self.training_stats)
256     self.display(1,"Rating statistics for test set: ",self.test_stats)

```

17.1.3 Ratings of top items and users

Sometimes it is useful to plot a property for all *(user,item,rating)* triples. There are too many such triples in the data set. The method *create_top_subset* creates a much smaller dataset where this makes sense. It picks the most rated items, then picks the users who have the most ratings on these items. It is designed for depicting the meaning of properties, and may not be useful for other purposes. The resulting plot is shown in Figure 17.3

```

_____relnCollFilt.py — (continued)_____
258 class Rating_set_top_subset(Rating_set):
259
260     def __init__(self, rating_set, num_items = (20,40), num_users =
        (20,24)):
261         """Returns a subset of the ratings by picking the most rated items,
262         and then the users that have most ratings on these, and then all of
        the
263         ratings that involve these users and items.
264         num_items is (ni,si) which selects ni users at random from the top
        si users
265         num_users is (nu,su) which selects nu items at random from the top
        su items
266         """
267         (ni, si) = num_items
268         (nu, su) = num_users
269         items = {item for (user,item,rating) in rating_set.training_data}

```

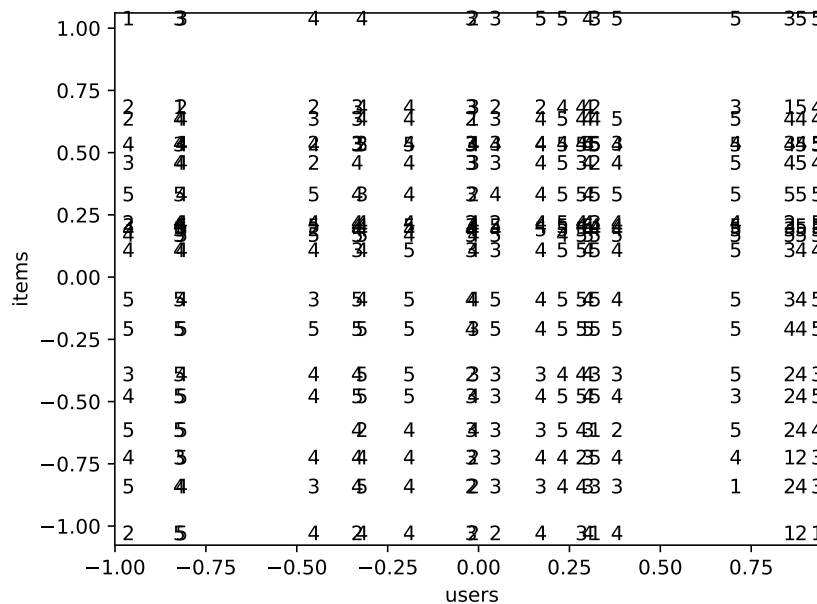


Figure 17.3: `learner1.plot_property(0)` for 20 most rated items and 20 users with most ratings on these. Users and items with similar property values overwrite each other.

```

270 item_counts = {i:0 for i in items}
271 for (user,item,rating) in rating_set.training_data:
272     item_counts[item] += 1
273
274 items_sorted = sorted((item_counts[i],i) for i in items)
275 top_items = random.sample([item for (count, item) in
276     items_sorted[-si:]], ni)
277 set_top_items = set(top_items)
278
279 users = {user for (user,item,rating) in rating_set.training_data}
280 user_counts = {u:0 for u in users}
281 for (user,item,rating) in rating_set.training_data:
282     if item in set_top_items:
283         user_counts[user] += 1
284
285 users_sorted = sorted((user_counts[u],u) for u in users)
286 top_users = random.sample([user for (count, user) in
287     users_sorted[-su:]], nu)
288 set_top_users = set(top_users)
289
290 self.training_data = [ (user,item,rating)
291     for (user,item,rating) in rating_set.training_data

```



```

290         if user in set_top_users and item in set_top_items]
291         self.test_data = []
292
293     movielens = Rating_set_from_file()
294     learner1 = CF_learner(movielens, num_properties = 1)
295     # learner10 = CF_learner(movielens, num_properties = 10)
296     # learner1.learn(50)
297     # learner1.plot_predictions(examples = "training")
298     # learner1.plot_predictions(examples = "test")
299     # learner1.plot_property(0)
300     # movielens_subset = Rating_set_top_subset(movielens,num_items = (20,40),
301         num_users = (20,40))
302     # learner_s = CF_learner(movielens_subset, num_properties=1)
303     # learner_s.learn(1000)
304     # learner_s.plot_property(0,plot_all=True)

```

17.2 Relational Probabilistic Models

The following implements relational belief networks – belief networks with plates. Plates correspond to logical variables.

```

_____relnProbModels.py — Relational Probabilistic Models: belief networks with plates_____
11 from display import Displayable
12 from probGraphicalModels import BeliefNetwork
13 from variable import Variable
14 from probRC import ProbRC
15 from probFactors import Prob
16 import random
17
18 boolean = [False, True]

```

A ParVar is a parametrized random variable, which consists of the name, a list of logical variables (plates), a domain, and a position. For each assignment of an entity to each logical variable, there is a random variable in a grounding.

```

_____relnProbModels.py — (continued) _____
20 class ParVar(object):
21     """Parametrized random variable"""
22     def __init__(self, name, log_vars, domain, position=None):
23         self.name = name # string
24         self.log_vars = log_vars
25         self.domain = domain # list of values
26         self.position = position if position else (random.random(),
27             random.random())
28         self.size = len(domain)

```

The class RBN is of relational belief networks. A relational belief network consists of a title, a set of parvariables, and a set of parfactors.

```

_____relnProbModels.py — (continued) _____

```

```

29 class RBN(Displayable):
30     def __init__(self, title, parvars, parfactors):
31         self.title = title
32         self.parvars = parvars
33         self.parfactors = parfactors
34         self.log_vars = {V for PV in parvars for V in PV.log_vars}

```

The grounding of a belief network with a population for each logical variable is a belief network, for which any of the belief network inference algorithms work.

```

relnProbModels.py — (continued)
36 def ground(self, populations, offsets=None):
37     """Ground the belief network with the populations of the logical
38         variables.
39         populations is a dictionary that maps each logical variable to the
40         list of individuals.
41         Returns a belief network representation of the grounding.
42     """
43     assert all(lv in populations for lv in self.log_vars), f"[[lv for
44         lv in self.log_vars if lv not in populations]] have no
45         population"
46     self.cps = [] # conditional probabilities in the grounding
47     self.var_dict = {} # ground variables created
48     for pp in self.parfactors:
49         self.ground_parfactor(pp, list(self.log_vars), populations, {},
50                               offsets)
51     return BeliefNetwork(self.title+"_grounded",
52                          self.var_dict.values(), self.cps)
53
54 def ground_parfactor(self, parfactor, lvs, populations, context,
55                     offsets):
56     """
57     parfactor is the parfactor to get instances of
58     lvs is a list of the logical variables in parfactor not assigned in
59     context
60     populations is {logical_variable: population} dictionary
61     context is a {logical_variable:value} dictionary for
62         logical_variable in parfactor
63     offsets a {loc_var:(x_offset,y_offset)} dictionary or None
64     """
65     if lvs == []:
66         if isinstance(parfactor, Prob):
67             self.cps.append(Prob(self.ground_pvr(parfactor.child, context, offsets),
68                                  [self.ground_pvr(p, context, offsets)
69                                   for p in parfactor.parents],
69                                  parfactor.values))
67         else:
68             print("Parfactor not implemented for", parfactor, "of
69                 type", type(parfactor))
70     else:

```

```

64         for val in populations[lvs[0]]:
65             self.ground_parfactor(parfactor, lvs[1:], populations,
                                   {lvs[0]:val}|context, offsets)
66
67     def ground_pvr(self, prv, context, offsets):
68         """grounds a parametrized random variable with respect to a context
69         prv is a parametrized random variable
70         context is a logical_variable:value dictionary that assigns all
71         logical variables in prv
72         offsets a {loc_var:(x_offset,y_offset)} dictionary or None
73         """
74         if isinstance(prv, ParVar):
75             args = tuple(context[lv] for lv in prv.log_vars)
76             if (prv, args) in self.var_dict:
77                 return self.var_dict[(prv, args)]
78             else:
79                 new_gv = GrVar(prv, args, offsets)
80                 self.var_dict[(prv, args)] = new_gv
81                 return new_gv
82         else: # allows for non-parametrized random variables
83             return prv

```

A GrVar is a variable constructed by grounding a parametrized random variable with respect to a tuple of values for the logical variables.

```

relnProbModels.py — (continued)
84 class GrVar(Variable):
85     """Grounded Variable"""
86     def __init__(self, parvar, args, offsets = None):
87         """A grounded variable
88         parvar is the parametrized variable
89         args is a tuple of a value for each random variable
90         offsets is a map between the value and the (x,y) offsets
91         """
92         if offsets:
93             pos = sum_positions([parvar.position]+[offsets[a] for a in
94             args])
95         else:
96             pos = sum_positions([parvar.position,
97             (random.uniform(-0.2,0.2),random.uniform(-0.2,0.2))])
98         Variable.__init__(self, parvar.name+"("+".join(args)+")",
99         parvar.domain, pos)
100         self.parvar= parvar
101         self.args = tuple(args)
102         self.hash_value = None
103
104     def __hash__(self):
105         if self.hash_value is None: # only hash once
106             self.hash_value = hash((self.parvar, self.args))
107         return self.hash_value

```

```

106     def __eq__(self, other):
107         return isinstance(other, GrVar) and self.parvar == other.parvar and
            self.args == other.args
108
109     def sum_positions(poslist):
110         (x,y) = (0,0)
111         for (xo,yo) in poslist:
112             x += xo
113             y += yo
114         return (x,y)

```

The following is a representation of Examples 17.5-17.7 of Poole and Mackworth [2023]. The plate model – represented here using grades – is shown in Figure 17.4. The observation in obs corresponds to the dataset of Figure 17.3. The grounding in grades_gr corresponds to Figure 17.5, but also includes the Grade variables not needed to answer the query (see exercise below).

Try the commented out queries to the Python shell:

```

relnProbModels.py — (continued)
116 Int = ParVar("Intelligent", ["St"], boolean, position=(0.0,0.7))
117 Grade = ParVar("Grade", ["St","Co"], ["A", "B", "C"], position=(0.2,0.6))
118 Diff = ParVar("Difficult", ["Co"], boolean, position=(0.3,0.9))
119
120 pg = Prob(Grade, [Int, Diff],
121           [{"A": 0.1, "B":0.4, "C":0.5},
122            {"A": 0.01, "B":0.09, "C":0.9}],
123           [{"A": 0.9, "B":0.09, "C":0.01},
124            {"A": 0.5, "B":0.4, "C":0.1}]))
125 pi = Prob( Int, [], [0.5, 0.5])
126 pd = Prob( Diff, [], [0.5, 0.5])
127 grades = RBN("Grades RBN", {Int, Grade, Diff}, {pg,pi,pd})
128
129 students = ["s1", "s2", "s3", "s4"]
130 st_offsets = {st:(0,-0.2*i) for (i,st) in enumerate(students)}
131 courses = ["c1", "c2", "c3", "c4"]
132 co_offsets = {co:(0.2*i,0) for (i,co) in enumerate(courses)}
133 grades_gr = grades.ground({"St": students, "Co": courses},
134                           offsets= st_offsets | co_offsets)
135
136 obs = {GrVar(Grade,["s1","c1"]):"A", GrVar(Grade,["s2","c1"]):"C",
137        GrVar(Grade,["s1","c2"]):"B",
138        GrVar(Grade,["s2","c3"]):"B", GrVar(Grade,["s3","c2"]):"B",
139        GrVar(Grade,["s4","c3"]):"B"}
140
141 # grades_rc = ProbRC(grades_gr)
142 # grades_rc.show_post({GrVar(Grade,["s1","c1"]):"A"}, fontsize=10)
143 #
144 #     grades_rc.show_post({GrVar(Grade,["s1","c1"]):"A",GrVar(Grade,["s2","c1"]):"C"})
145 #
146 #     grades_rc.show_post({GrVar(Grade,["s1","c1"]):"A",GrVar(Grade,["s2","c1"]):"C",
147 #                          GrVar(Grade,["s1","c2"]):"B"})

```

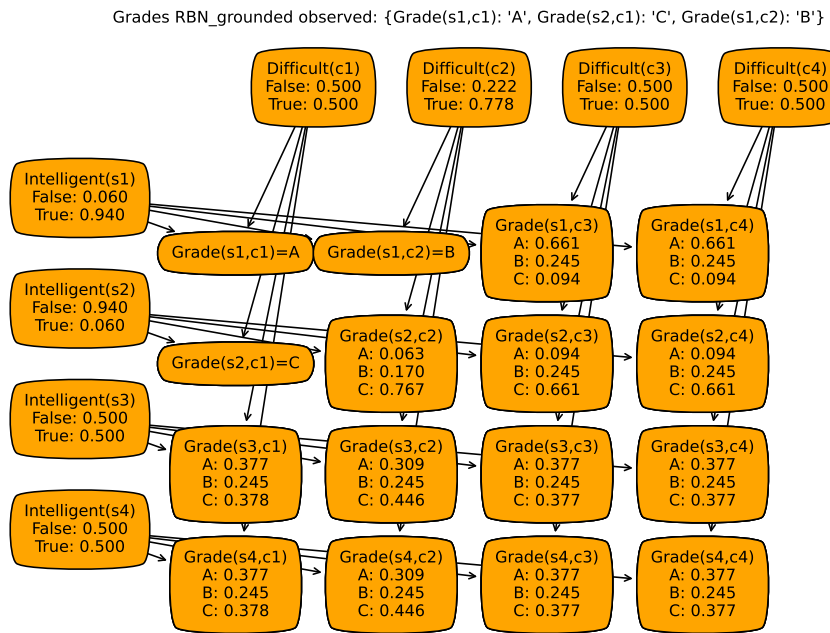


Figure 17.4: Grounded network with three observations

```

143 # grades_rc.show_post(obs, fontsize=10)
144 # grades_rc.query(GrVar(Grade, ["s3", "c4"]), obs)
145 # grades_rc.query(GrVar(Grade, ["s4", "c4"]), obs)
146 # grades_rc.query(GrVar(Int, ["s3"]), obs)
147 # grades_rc.query(GrVar(Int, ["s4"]), obs)

```

Figure 17.4 shows the distribution over ground variables after the 3rd `show_post` in the code above (with 3 grades observed).

Exercise 17.5 What are advantages and disadvantages of using this formulation over using `CF_learner` with `grades_rs`? Think about overfitting, and where the parameters come from.

Exercise 17.6 The grounding above creates a random variable for each element for each possible combination of individuals in the populations. Change it so that it only creates as many random variables as needed to answer a query. For example, for the observations and queries above, only the variables in Figure 17.5 in Poole and Mackworth [2023] need to be created.

Version History

- 2024-12-09 Version 0.9.14. Code simplified, user manual has more explanation. This is candidate release for Version 1.0.
- 2024-04-30 Version 0.9.13: Minor changes including counterfactual reasoning.
- 2023-12-06 Version 0.9.12: Top-down proof for Datalog (ch 15) and triple store (ch 16)
- 2023-11-21 Version 0.9.11 updated and simplified relational learning, show relational belief networks
- 2023-11-07 Version 0.9.10 Improved GUIs and test cases for decision-theoretic planning (MDPs) and reinforcement learning.
- 2023-10-6 Version 0.9.8 GUIs for search, Bayesian learning, causality and many smaller changes.
- 2023-07-31 Version 0.9.7 includes relational probabilistic models and smaller changes
- 2023-06-06 Version 0.9.6 controllers are more consistent. Many smaller changes.
- 2022-08-13 Version 0.9.5 major revisions including extra code for causality and deep learning
- 2021-07-08 Version 0.9.1 updated the CSP code to have the same representation of variables as used by the probability code

- 2021-05-13 Version 0.9.0 Major revisions to chapters 8 and 9. Introduced recursive conditioning, simplified much code. New section on 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 fixed arc costs.
- 2020-07-21 Version 0.8.2 added positions and string to constraints
- 2019-09-17 Version 0.8.0 represented blocks world (Section 6.1.2) due to bug found by Donato Meoli.

Bibliography

- Chen, T. and Guestrin, C. (2016), Xgboost: A scalable tree boosting system. In *KDD '16: 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, URL <https://doi.org/10.1145/2939672.2939785>. 187
- Chollet, F. (2021), *Deep Learning with Python*. Manning. 189
- Dua, D. and Graff, C. (2017), UCI machine learning repository. URL <http://archive.ics.uci.edu/ml>. 151
- 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>. 190
- Harper, F. M. and Konstan, J. A. (2015), The MovieLens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems*, 5(4). 390
- 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*. 187
- Koren, Y., Bell, R., and Volinsky, C. (2009), Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37. 383
- Lichman, M. (2013), UCI machine learning repository. URL <http://archive.ics.uci.edu/ml>. 151
- Pearl, J. (2009), *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2nd edition. 215, 280

Pérez, F. and Granger, B. E. (2007), IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, URL <https://ipython.org>. 10

Poole, D. L. and Mackworth, A. K. (2023), *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 3rd edition, URL <https://artint.info>. 9, 25, 27, 39, 40, 48, 50, 51, 77, 116, 125, 196, 210, 213, 214, 221, 222, 265, 298, 301, 303, 304, 321, 323, 329, 334, 336, 354, 362, 369, 370, 372, 374, 380, 383, 396, 397

Index

- α - β pruning, 358
- A* search, 54
- A* Search, 61
- action, 127
- agent, 25, 317
- argmax, 19
- assignment, 72, 203
- assumable, 121
- asynchronous value iteration, 315
- augmented feature, 163
- Bayesian network, 210
- belief network, 210
- blocks world, 130
- Boolean feature, 152
- bottom-up proof, 114
- branch-and-bound search, 65
- class
 - Action_instance*, 144
 - Agent*, 26
 - Arc*, 42
 - Askable*, 111
 - Assumable*, 121
 - BNfromDBN*, 255
 - BeliefNetwork*, 211
 - Boosted_dataset*, 184
 - Boosting_learner*, 185
 - Branch_and_bound*, 108
 - CF_learner*, 383
 - CPD*, 205
 - CPDrename*, 252
 - CSP*, 73
 - CSP_from_STRIPS*, 140
 - Clause*, 111, 368
 - Con_solver*, 89
 - ConstantCPD*, 205
 - Constraint*, 72
 - DBN*, 252
 - DBNVEfilter*, 256
 - DBNvariable*, 251
 - DF_Branch_and_bound*, 65
 - DT_learner*, 169
 - Data_from_file*, 160
 - Data_from_files*, 161
 - Data_set*, 153
 - Data_set_augmented*, 163
 - DecisionFunction*, 291
 - DecisionNetwork*, 284
 - DecisionVariable*, 284
 - Displayable*, 18
 - Dist*, 209

- Dropout_layer*, 196
- EM_learner*, 268
- Env_from_MDP*, 322
- Environment*, 26
- Evaluate*, 158
- Factor*, 203
- FactorMax*, 296
- FactorObserved*, 227
- FactorRename*, 252
- FactorSum*, 227
- Forward_STRIPS*, 133
- FrontierPQ*, 60
- GTB_learner*, 187
- GibbsSampling*, 236
- GrVar*, 395
- GraphicalModel*, 210
- GridDomain*, 310
- HMM*, 241
- HMMVEfilter*, 242
- HMM_Controlled*, 244
- HMM_Local*, 245
- HMMparticleFilter*, 247
- IFeq*, 208
- InferenceMethod*, 218, 273
- KB*, 112, 370
- KBA*, 121
- KBT*, 380
- K_fold_dataset*, 174
- K_means_learner*, 263
- Layer*, 189
- Learner*, 165
- LikelihoodWeighting*, 233
- Linear_complete_layer*, 191
- Linear_complete_layer_RMS_Prop*, 195
- Linear_complete_layer_momentum*, 195
- Linear_learner*, 176
- LogisticRegression*, 206
- MDP*, 298
- MDPtiny*, 302
- Magic_sum*, 355
- Model_based_reinforcement_learner*, 336
- Monster_game_env*, 305, 323
- NN*, 193
- Node*, 354
- NoisyOR*, 206
- POP_node*, 145
- POP_search_from_STRIPS*, 146
- ParVar*, 393
- ParticleFiltering*, 234
- Party_env*, 321
- Path*, 45
- Planning_problem*, 128
- Plot_env*, 36
- Plot_prices*, 29
- Predict*, 166
- Prob*, 208
- ProbDT*, 208
- ProbRC*, 222
- ProbSearch*, 221
- Q_learner*, 326
- RBN*, 393
- RC_DN*, 292
- RL_agent*, 318
- RL_env*, 317
- Rating_set*, 390
- ReLU_layer*, 192
- Regression_STRIPS*, 137
- RejectionSampling*, 232
- Rob_body*, 32
- Rob_env*, 31
- Rob_middle_layer*, 34
- Rob_top_layer*, 35
- Runtime_distribution*, 105
- SARSA*, 328
- SARSA_LFA_learner*, 342
- SLSearcher*, 98
- STRIPS_domain*, 128
- SameAs*, 209
- SamplingInferenceMethod*, 231
- Search_from_CSP*, 85, 87
- Search_problem*, 41
- Search_problem_from_explicit_graph*, 43
- Search_with_AC_from_CSP*, 96
- Searcher*, 54
- SearcherGUI*, 56
- SearcherMPP*, 63

- Show_Localization*, 245
- Sigmoid_layer*, 192
- SoftConstraint*, 107
- State*, 133
- Strips*, 127
- Subgoal*, 137
- TP_agent*, 28
- TP_env*, 27
- TabFactor*, 207
- TripleStore*, 377
- Updatable_priority_queue*, 103
- Utility*, 283
- UtilityTable*, 283
- VE*, 226
- VE_DN*, 296
- Variable*, 71
- clause, 111
- collaborative filtering, 383
- comprehensions, 12
- condition, 72
- conditional probability distribution
 - (CPD), 205
- consistency algorithms, 89
- constraint, 72
- constraint satisfaction problem, 71
- CPD (conditional probability distribution), 205
- cross validation, 174
- CSP, 71
 - consistency, 89
 - domain splitting, 91, 96
 - search, 87
 - stochastic local search, 97
- currying, 76
- datalog, 367
- dataset, 152
- DBN
 - filtering, 256
 - unrolling, 255
- DBN (dynamic belief network), 250
- debugging, 117
- decision network, 283
- decision tree learning, 169
- decision tree factors, 208
- decision variable, 283
- deep learning, 189
- display, 18
- Displayable, 18
- domain splitting, 91, 96
- Dropout, 196
- dynamic belief network (DBN), 250
 - representation, 250
- EM, 268
- environment, 25, 26, 317
- error, 157
- example, 152
- explanation, 117
- explicit graph, 43
- factor, 203, 207
- factor_times, 228
- feature, 152, 154
- feature engineering, 151
- file
 - agentBuying.py*, 27
 - agentEnv.py*, 31
 - agentFollowTarget.py*, 39
 - agentMiddle.py*, 34
 - agentTop.py*, 35
 - agents.py*, 26
 - cspConsistency.py*, 89
 - cspConsistencyGUI.py*, 94
 - cspDFS.py*, 85
 - cspExamples.py*, 76
 - cspProblem.py*, 72
 - cspSLS.py*, 98
 - cspSearch.py*, 87
 - cspSoft.py*, 107
 - decnNetworks.py*, 283
 - display.py*, 18
 - knowledgeGraph.py*, 377
 - knowledgeReasoning.py*, 380
 - learnBayesian.py*, 259
 - learnBoosting.py*, 184
 - learnCrossValidation.py*, 174
 - learnDT.py*, 169
 - learnEM.py*, 268
 - learnKMeans.py*, 263

- learnLinear.py*, 176
- learnNN.py*, 189
- learnNoInputs.py*, 166
- learnProblem.py*, 152
- logicAssumables.py*, 121
- logicBottomUp.py*, 114
- logicExplain.py*, 117
- logicNegation.py*, 124
- logicProblem.py*, 111
- logicRelation.py*, 367
- logicTopDown.py*, 116
- masLearn.py*, 359
- masMiniMax.py*, 357
- masProblem.py*, 354
- mdpExamples.py*, 298
- mdpGUI.py*, 310
- mdpProblem.py*, 298
- probCounterfactual.py*, 276
- probDBN.py*, 251
- probDo.py*, 273
- probExamples.py*, 213
- probFactors.py*, 203
- probGraphicalModels.py*, 210
- probHMM.py*, 241
- probLocalization.py*, 244
- probRC.py*, 221
- probStochSim.py*, 230
- probVE.py*, 226
- pythonDemo.py*, 13
- relnCollFilt.py*, 383
- relnExamples.py*, 370
- relnProbModels.py*, 393
- rlExamples.py*, 321
- rlFeatures.py*, 342
- rlGUI.py*, 346
- rlGameFeature.py*, 339
- rlModelLearner.py*, 336
- rlProblem.py*, 317
- rlQExperienceReplay.py*, 331
- rlQLearner.py*, 326
- rlStochasticPolicy.py*, 334
- searchBranchAndBound.py*, 65
- searchExample.py*, 47
- searchGUI.py*, 56
- searchGeneric.py*, 54
- searchGrid.py*, 64
- searchMPP.py*, 63
- searchProblem.py*, 41
- searchTest.py*, 67
- stripsCSPPlanner.py*, 140
- stripsForwardPlanner.py*, 133
- stripsHeuristic.py*, 135
- stripsPOP.py*, 144
- stripsProblem.py*, 127
- stripsRegressionPlanner.py*, 137
- utilities.py*, 19
- variable.py*, 71
- filtering, 242, 247
 - DBN, 256
- flip, 20
- forward planning, 132
- frange, 154
- ftype, 154
- fully observable, 317
- game, 353
- Gibbs sampling, 236
- graphical model, 210
- heuristic planning, 135, 139
- hidden Markov model, 241
- hierarchical controller, 31
- HMM
 - exact filtering, 242
 - particle filtering, 247
- HMM (hidden Markov models), 241
- importance sampling, 234
- interact
 - proofs, 118
- ipython, 10
- k-means, 263
- kernel, 163
- knowledge base, 112
- knowledge graph, 377
- learner, 165
- learning, 151–201, 259–272, 317–351, 383–393
 - cross validation, 174

- decision tree, 169
- deep, 189–201
- deep learning, 189
- EM, 268
- k-means, 263
- linear regression, 176
- linear classification, 176
- neural network, 189
- no inputs, 165
- reinforcement, 317–351
- relational, 383
- supervised, 151–188
- with uncertainty, 259–272
- LightGBM, 187
- likelihood weighting, 233
- linear regression, 176
- linear classification, 176
- localization, 244
- logic program, 367
- logistic regression, 206
- logit, 179
- loss, 157
- magic square, 355
- magic-sum game, 355
- Markov Chain Monte Carlo, 236
- Markov decision process, 298
- max_display_level, 18
- MCMC, 236
- MDP, 298, 322
 - GUI, 310
- method
 - consistent*, 74
 - holds*, 73
 - maxh*, 135
 - zero*, 133
- minimax, 353
- minimax algorithm, 357
- minsets, 122
- model-based reinforcement learner, 336
- multiagent system, 353
- multiple path pruning, 63
- n-queens problem, 84
- naive search probabilistic inference, 220
- naughts and crosses, 355
- neural network, 189
- noisy-or, 206
- NotImplementedError, 26
- partial-order planner, 144
- particle filtering, 234
 - HMMs, 247
- planning, 127–150, 283–316
 - CSP, 140
 - decision network, 283
 - forward, 132
 - MDP, 298
 - partial order, 144
 - regression, 137
 - with certainty, 127–150
 - with learning, 336
 - with uncertainty, 283–316
- plotting
 - agents in time, 29
 - reinforcement learning, 320
 - robot environment, 36
 - run-time distribution, 105
 - stochastic simulation, 238
- predictor, 157
- Prob, 208
- probabilistic inference methods, 218
- probability, 203
- proof
 - bottom-up, 114
 - explanation, 117
 - top-down, 116, 372
- proposition, 111
- Python, 9
- Q learning, 326
- query, 218
- queryD0, 273
- RC, 222, 292
- recursive conditioning (RC), 222
- recursive conditioning for decision networks, 292
- regression planning, 137

- reinforcement learning, 317–351
 - environment, 317
 - feature-based, 339
 - model-based, 336
 - Q-learning, 326
- rejection sampling, 232
- relational learning, 383
- relations, 367
- ReLU, 192
- resampling, 235
- robot
 - body, 31
 - middle layer, 34
 - plotting, 36
 - top layer, 35
 - world, 31
- robot delivery domain, 128
- run time, 16
- runtime distribution, 105
- sampling, 230
 - importance sampling, 234
 - belief networks, 231
 - likelihood weighting, 233
 - particle filtering, 234
 - rejection, 232
- SARSA, 328
- scope, 72
- search, 41
 - A*, 54
 - branch-and-bound, 65
 - multiple path pruning, 63
- search_with_any_conflict, 100
- search_with_var_pq, 101
- show, 74, 212
- sigmoid, 179
- softmax, 179
- stochastic local search, 97
 - any-conflict, 100
 - two-stage choice, 101
- stochastic simulation, 230
- tabular factor, 207
- test
 - SLS, 106
- tic-tac-toe, 355
- top-down proof, 116, 372
- triple store, 377, 380
- uncertainty, 203
- unification, 369, 370
- unit test, 21, 61, 85, 115, 116, 118
- unrolling
 - DBN, 255
- updatable priority queue, 103
- utility, 283
- utility table, 283
- value iteration, 308
- variable, 71
- variable elimination (VE), 226
- variable elimination for decision networks, 295
- VE, 226
- XGBoost, 187
- yield, 13