

# Python code for Artificial Intelligence Foundations of Computational Agents

David L. Poole and Alan K. Mackworth

Version 0.9.15 of April 11, 2025.

<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 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

<b>Contents</b>	<b>3</b>
<b>1 Python for Artificial Intelligence</b>	<b>9</b>
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
<b>2 Agent Architectures and Hierarchical Control</b>	<b>25</b>
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
<b>3</b>	<b>Searching for Solutions</b>	<b>41</b>
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 . . . . .	60
3.2.4	$A^*$ Search . . . . .	61
3.2.5	Multiple Path Pruning . . . . .	63
3.3	Branch-and-bound Search . . . . .	65
<b>4</b>	<b>Reasoning with Constraints</b>	<b>69</b>
4.1	Constraint Satisfaction Problems . . . . .	69
4.1.1	Variables . . . . .	69
4.1.2	Constraints . . . . .	70
4.1.3	CSPs . . . . .	71
4.1.4	Examples . . . . .	74
4.2	A Simple Depth-first Solver . . . . .	83
4.3	Converting CSPs to Search Problems . . . . .	85
4.4	Consistency Algorithms . . . . .	87
4.4.1	Direct Implementation of Domain Splitting . . . . .	89
4.4.2	Consistency GUI . . . . .	91
4.4.3	Domain Splitting as an interface to graph searching . . . . .	94
4.5	Solving CSPs using Stochastic Local Search . . . . .	96
4.5.1	Any-conflict . . . . .	98
4.5.2	Two-Stage Choice . . . . .	99
4.5.3	Updatable Priority Queues . . . . .	101
4.5.4	Plotting Run-Time Distributions . . . . .	103
4.5.5	Testing . . . . .	104
4.6	Discrete Optimization . . . . .	105
4.6.1	Branch-and-bound Search . . . . .	106
<b>5</b>	<b>Propositions and Inference</b>	<b>109</b>
5.1	Representing Knowledge Bases . . . . .	109
5.2	Bottom-up Proofs (with askables) . . . . .	112

5.3	Top-down Proofs (with askables) . . . . .	114
5.4	Debugging and Explanation . . . . .	115
5.5	Assumables . . . . .	119
5.6	Negation-as-failure . . . . .	122
<b>6</b>	<b>Deterministic Planning</b>	<b>125</b>
6.1	Representing Actions and Planning Problems . . . . .	125
6.1.1	Robot Delivery Domain . . . . .	126
6.1.2	Blocks World . . . . .	128
6.2	Forward Planning . . . . .	130
6.2.1	Defining Heuristics for a Planner . . . . .	133
6.3	Regression Planning . . . . .	135
6.3.1	Defining Heuristics for a Regression Planner . . . . .	137
6.4	Planning as a CSP . . . . .	138
6.5	Partial-Order Planning . . . . .	142
<b>7</b>	<b>Supervised Machine Learning</b>	<b>149</b>
7.1	Representations of Data and Predictions . . . . .	150
7.1.1	Creating Boolean Conditions from Features . . . . .	153
7.1.2	Evaluating Predictions . . . . .	155
7.1.3	Creating Test and Training Sets . . . . .	157
7.1.4	Importing Data From File . . . . .	158
7.1.5	Augmented Features . . . . .	161
7.2	Generic Learner Interface . . . . .	163
7.3	Learning With No Input Features . . . . .	163
7.3.1	Evaluation . . . . .	166
7.4	Decision Tree Learning . . . . .	167
7.5	Cross Validation and Parameter Tuning . . . . .	172
7.6	Linear Regression and Classification . . . . .	175
7.7	Boosting . . . . .	182
7.7.1	Gradient Tree Boosting . . . . .	185
<b>8</b>	<b>Neural Networks and Deep Learning</b>	<b>187</b>
8.1	Layers . . . . .	187
8.1.1	Linear Layer . . . . .	188
8.1.2	Stochastic Gradient Descent . . . . .	190
8.1.3	ReLU Layer . . . . .	190
8.1.4	Sigmoid Layer . . . . .	191
8.2	Feedforward Networks . . . . .	191
8.3	Alternative Optimizers . . . . .	193
8.3.1	Momentum . . . . .	193
8.3.2	RMS-Prop . . . . .	194
8.4	Dropout . . . . .	195
8.5	Examples . . . . .	196
8.6	Plotting Performance . . . . .	197

<b>9</b>	<b>Reasoning with Uncertainty</b>	<b>201</b>
9.1	Representing Probabilistic Models . . . . .	201
9.2	Representing Factors . . . . .	201
9.3	Conditional Probability Distributions . . . . .	203
9.3.1	Logistic Regression . . . . .	204
9.3.2	Noisy-or . . . . .	204
9.3.3	Tabular Factors and Prob . . . . .	205
9.3.4	Decision Tree Representations of Factors . . . . .	206
9.4	Graphical Models . . . . .	208
9.4.1	Showing Belief Networks . . . . .	210
9.4.2	Example Belief Networks . . . . .	210
9.5	Inference Methods . . . . .	216
9.5.1	Showing Posterior Distributions . . . . .	217
9.6	Naive Search . . . . .	218
9.7	Recursive Conditioning . . . . .	220
9.8	Variable Elimination . . . . .	224
9.9	Stochastic Simulation . . . . .	228
9.9.1	Sampling from a discrete distribution . . . . .	228
9.9.2	Sampling Methods for Belief Network Inference . . . . .	230
9.9.3	Rejection Sampling . . . . .	230
9.9.4	Likelihood Weighting . . . . .	231
9.9.5	Particle Filtering . . . . .	232
9.9.6	Examples . . . . .	234
9.9.7	Gibbs Sampling . . . . .	235
9.9.8	Plotting Behavior of Stochastic Simulators . . . . .	236
9.10	Hidden Markov Models . . . . .	239
9.10.1	Exact Filtering for HMMs . . . . .	241
9.10.2	Localization . . . . .	242
9.10.3	Particle Filtering for HMMs . . . . .	245
9.10.4	Generating Examples . . . . .	247
9.11	Dynamic Belief Networks . . . . .	248
9.11.1	Representing Dynamic Belief Networks . . . . .	249
9.11.2	Unrolling DBNs . . . . .	253
9.11.3	DBN Filtering . . . . .	255
<b>10</b>	<b>Learning with Uncertainty</b>	<b>257</b>
10.1	Bayesian Learning . . . . .	257
10.2	K-means . . . . .	261
10.3	EM . . . . .	266
<b>11</b>	<b>Causality</b>	<b>271</b>
11.1	Do Questions . . . . .	271
11.2	Counterfactual Reasoning . . . . .	274
11.2.1	Choosing Deterministic System . . . . .	274
11.2.2	Firing Squad Example . . . . .	278

<b>12 Planning with Uncertainty</b>	<b>281</b>
12.1 Decision Networks . . . . .	281
12.1.1 Example Decision Networks . . . . .	283
12.1.2 Decision Functions . . . . .	289
12.1.3 Recursive Conditioning for Decision Networks . . . . .	290
12.1.4 Variable elimination for decision networks . . . . .	293
12.2 Markov Decision Processes . . . . .	296
12.2.1 Problem Domains . . . . .	297
12.2.2 Value Iteration . . . . .	306
12.2.3 Value Iteration GUI for Grid Domains . . . . .	307
12.2.4 Asynchronous Value Iteration . . . . .	311
<b>13 Reinforcement Learning</b>	<b>315</b>
13.1 Representing Agents and Environments . . . . .	315
13.1.1 Environments . . . . .	315
13.1.2 Agents . . . . .	316
13.1.3 Simulating an Environment-Agent Interaction . . . . .	317
13.1.4 Party Environment . . . . .	319
13.1.5 Environment from a Problem Domain . . . . .	320
13.1.6 Monster Game Environment . . . . .	321
13.2 Q Learning . . . . .	324
13.2.1 Exploration Strategies . . . . .	327
13.2.2 Testing Q-learning . . . . .	327
13.3 Q-learning with Experience Replay . . . . .	329
13.4 Stochastic Policy Learning Agent . . . . .	332
13.5 Model-based Reinforcement Learner . . . . .	334
13.6 Reinforcement Learning with Features . . . . .	337
13.6.1 Representing Features . . . . .	337
13.6.2 Feature-based RL learner . . . . .	340
13.7 GUI for RL . . . . .	343
<b>14 Multiagent Systems</b>	<b>351</b>
14.1 Minimax . . . . .	351
14.1.1 Creating a two-player game . . . . .	352
14.1.2 Minimax and $\alpha$ - $\beta$ Pruning . . . . .	355
14.2 Multiagent Learning . . . . .	357
14.2.1 Simulating Multiagent Interaction with an Environment . . . . .	357
14.2.2 Example Games . . . . .	360
14.2.3 Testing Games and Environments . . . . .	362
<b>15 Individuals and Relations</b>	<b>365</b>
15.1 Representing Datalog and Logic Programs . . . . .	365
15.2 Unification . . . . .	367
15.3 Knowledge Bases . . . . .	368
15.4 Top-down Proof Procedure . . . . .	370

15.5	Logic Program Example . . . . .	372
<b>16</b>	<b>Knowledge Graphs and Ontologies</b>	<b>375</b>
16.1	Triple Store . . . . .	375
16.2	Integrating Datalog and Triple Store . . . . .	378
<b>17</b>	<b>Relational Learning</b>	<b>381</b>
17.1	Collaborative Filtering . . . . .	381
17.1.1	Plotting . . . . .	385
17.1.2	Loading Rating Sets from Files and Websites . . . . .	388
17.1.3	Ratings of top items and users . . . . .	389
17.2	Relational Probabilistic Models . . . . .	391
<b>18</b>	<b>Version History</b>	<b>397</b>
	<b>Bibliography</b>	<b>399</b>
	<b>Index</b>	<b>401</b>



## 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:

```
pip 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:

```
pip 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 ipython or python (or perhaps ipython3 or python3) 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 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:<sup>1</sup>

```
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)))
```

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

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 |
```



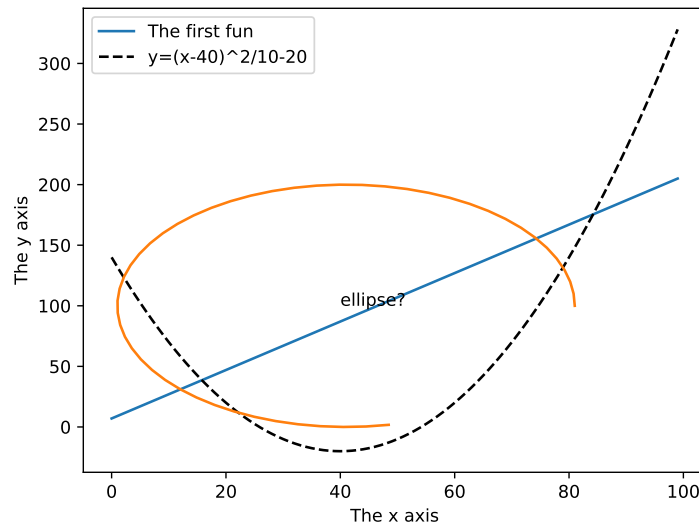


Figure 1.1: Result of pythonDemo code

```

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.

```

_____utilities.py — AIPython useful utilities_____
11 import random
12 import math
13
14 def argmaxall(gen):
15     """gen is a generator of (element,value) pairs, where value is a real.
16     argmaxall returns a list of all of the elements with maximal value.
17     """
18     maxv = -math.inf      # negative infinity
19     maxvals = []         # list of maximal elements
20     for (e,v) in gen:
21         if v > maxv:
22             maxvals, maxv = [e], v
23         elif v == maxv:
24             maxvals.append(e)
25     return maxvals
26
27 def argmaxe(gen):
28     """gen is a generator of (element,value) pairs, where value is a real.
29     argmaxe returns an element with maximal value.
30     If there are multiple elements with the max value, one is returned at
31         random.
32     """
33     return random.choice(argmaxall(gen))

```

```

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 # argmaxd({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](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 Ontologies
158     print("\n***** testing Knowledge Graphs and Ontologies *****")
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 315).
- 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

```

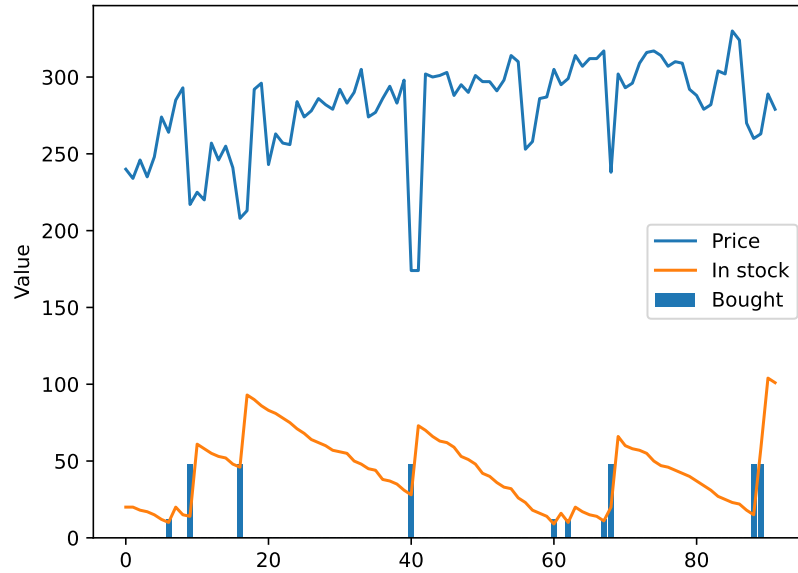


Figure 2.1: Percept and command traces for the paper-buying agent

```

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 =
            {'left':1,'straight':0,'right':-1}[direction]*self.turning_angle
63         self.rob_dir = (self.rob_dir + compass_deriv +360)%360 # make in
64         range [0,360)
65         rob_x_new = self.rob_x + math.cos(self.rob_dir*math.pi/180)
66         rob_y_new = self.rob_y + math.sin(self.rob_dir*math.pi/180)
67         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)

```

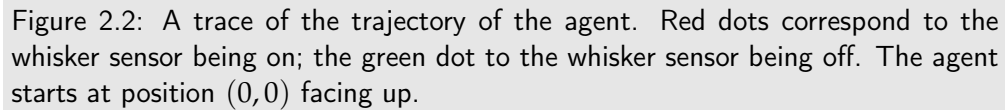
### 2.3.5 Plotting

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

```



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)

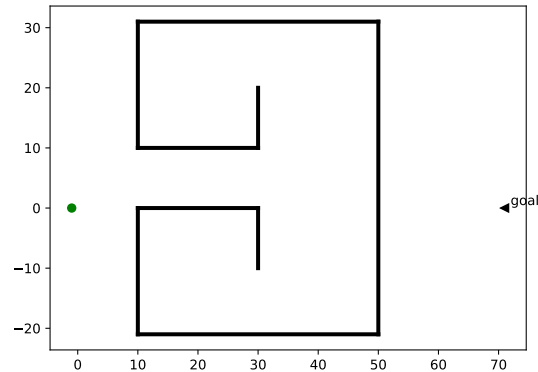


Figure 2.3: Robot trap

```

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?
88 |
89 | if __name__ == "__main__":
90 |     print("Try: Plot_env(body,top);
          top.do({'visit':['o109','storage','o109','o103']})")

```

**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)
92 | # Robot Trap for which the current controller cannot escape:
93 | trap_env = Rob_world({((10,-21),(10,0)), ((10,10),(10,31)),
94 |                      ((30,-10),(30,0)), ((30,10),(30,20)),
95 |                      ((50,-21),(50,31)), ((10,-21),(50,-21)),
96 |                      ((10,0),(30,0)), ((10,10),(30,10)),
97 |                      ((10,31),(50,31))})
98 | trap_body = Rob_body(trap_env,init_pos=(-1,0,90))
99 | trap_middle = Rob_middle_layer(trap_body)
100 | trap_top = Rob_top_layer(trap_middle,locations={'goal':(71,0)})
101 |

```

```

102 # Robot trap exercise:
103 # pl=Plot_env(trap_body,trap_top)
104 # 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)}")

```

```

45     self.pressloc = None
46     self.pressevent = None
47
48     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
57 # try:
58 # pl=Plot_follow(body,top)
59 # top.do({'visit':['o109','storage','o109','o103']})
60
61 if __name__ == "__main__":
62     print("Try: Plot_follow(body,top);
63         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}"

```

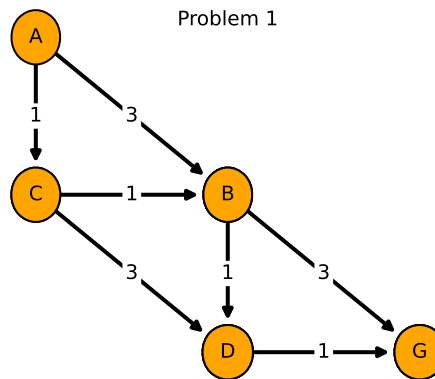


Figure 3.1: problem1

### 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)})

```

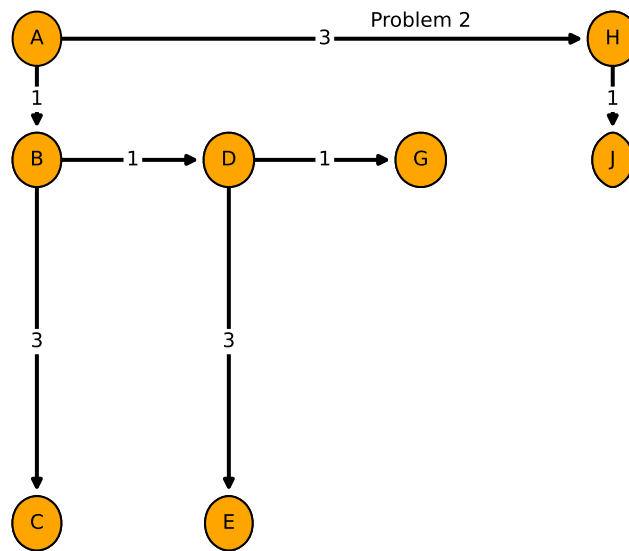


Figure 3.2: problem2

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.

searchExample.py — (continued)

```

31 problem3 = Search_problem_from_explicit_graph('Problem 3',
32     {'a','b','c','d','e','g','h','j'},
33     [],
34     start = 'g',
35     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].

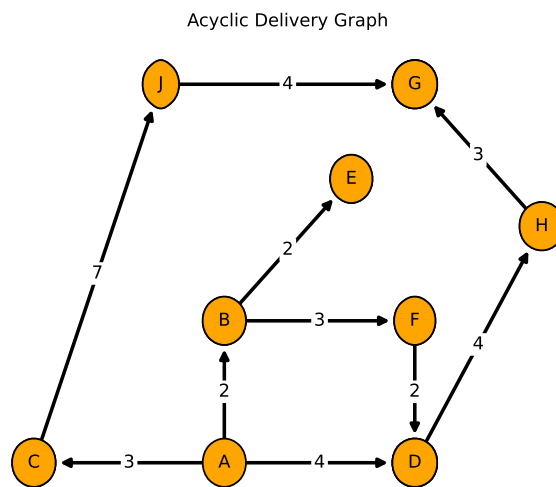
searchExample.py — (continued)

```

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

```

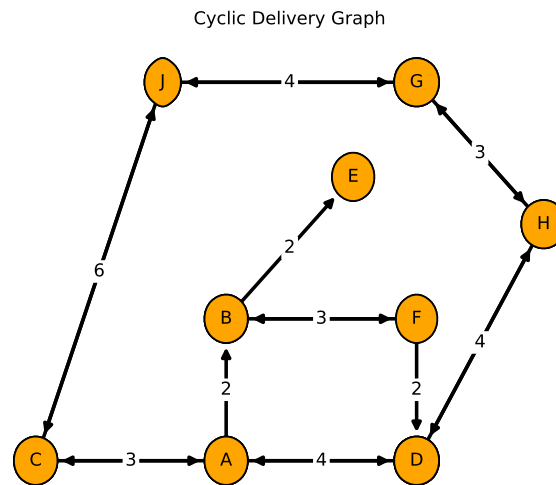


Figure 3.3: `simp_delivery_graph.show()`

```

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),
          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    })

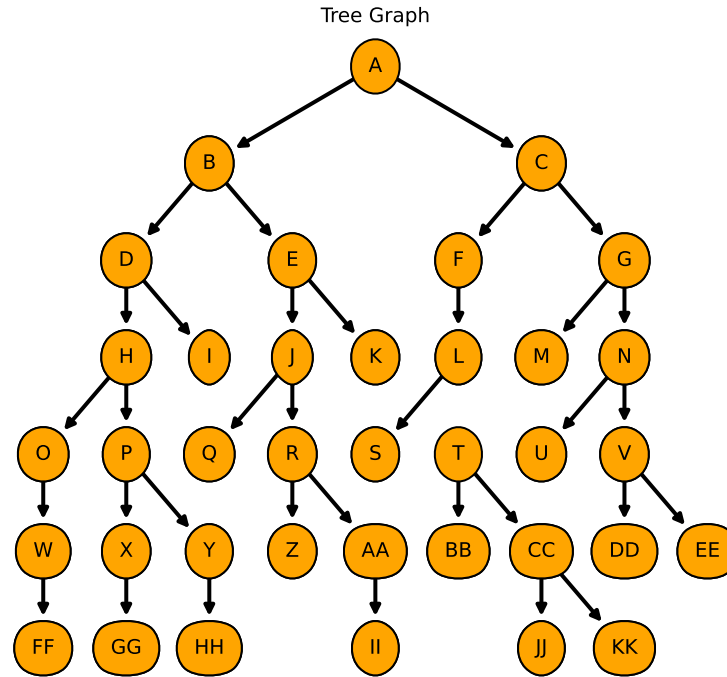
```

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     [ Arc('A', 'B', 1),
125       Arc('A', 'C', 1),
126       Arc('B', 'D', 1),
127       Arc('B', 'E', 1),
128       Arc('C', 'F', 1),
129       Arc('C', 'G', 1),
130       Arc('D', 'H', 1),
131       Arc('D', 'I', 1),
132       Arc('E', 'J', 1),
133       Arc('E', 'K', 1),
134       Arc('F', 'L', 1),
135       Arc('G', 'M', 1),
136       Arc('G', 'N', 1),
137       Arc('H', 'O', 1),
138       Arc('H', 'P', 1),
139       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

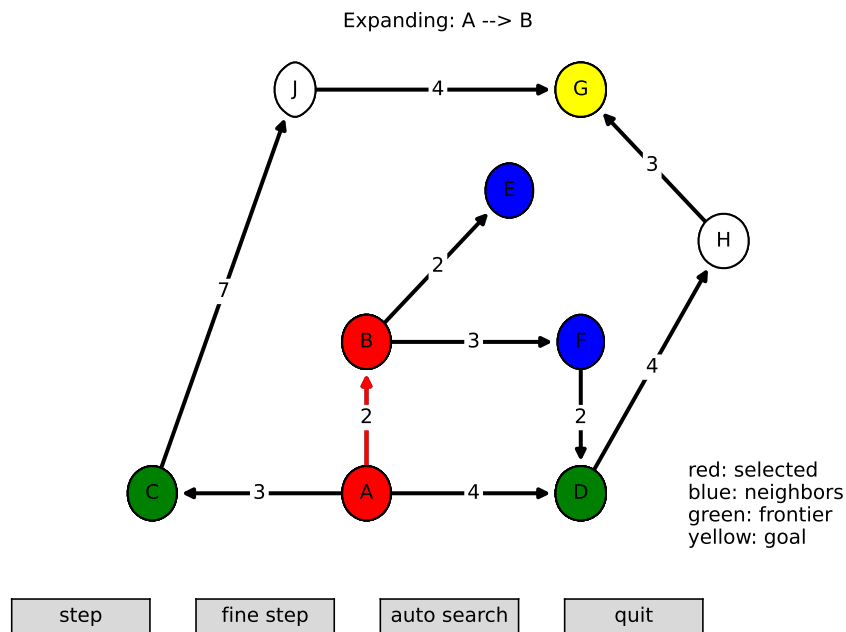


Figure 3.6: SearcherGUI(Searcher, simp\_delivery\_graph)

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([0.1,0.02,0.2,0.05]), "step")
33         step_but.on_clicked(self.step)
34         fine_but = Button(plt.axes([0.4,0.02,0.2,0.05]), "fine step")
35         fine_but.on_clicked(self.finestep)
36         auto_but = Button(plt.axes([0.7,0.02,0.2,0.05]), "auto search")
37         auto_but.on_clicked(self.auto)
38         fig.canvas.mpl_connect('close_event', self.window_closed)
39         self.ax.text(0.85,0, '\n'.join(self.colors[a] + ": " + a
40                                     for a in self.colors))
41         self.problem.show_graph(self.ax, node_color='white')
42         self.problem.show_node(self.ax, self.problem.start,
43                               self.colors['frontier'])
44         for node in self.problem.nodes:
45             if self.problem.is_goal(node):
46                 self.problem.show_node(self.ax, node, self.colors['goal'])
47         plt.show()
48         self.click = 7 # bigger than any display!
49         self.searcher.display = self.display
50         try:
51             while self.searcher.frontier:
52                 path = self.searcher.search()
53         except ExitToPython:
54             print("GUI closed")
55         else:
56             print("No more solutions")
57
58     def display(self, level, *args, **nargs):
59         if self.quitting:

```

```

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

```

```

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

```

searchGUI.py — (continued)

```

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

```

searchGeneric.py — (continued)
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,
143         solutions=[['G','D','B','C','A']] ):
144     """Unit test for aipython searching algorithms.
145     SearchClass is a class that takes a problem and implements search()
146     problem is a search problem
147     solutions is a list of optimal solutions
148     """
149     print("Testing problem 1:")
150     schr1 = SearchClass(problem)
151     path1 = schr1.search()
152     print("Path found:",path1)
153     assert path1 is not None, "No path is found in problem1"
154     assert list(path1.nodes()) in solutions, "Shortest path not found in
155         problem1"
156     print("Passed unit test")
157
158 if __name__ == "__main__":
159     #test(Searcher)      # what needs to be changed to make this succeed?
160     test(AStarSearcher)
161
162 # example queries:
163 # searcher1 = Searcher(searchExample.simp_delivery_graph) # DFS
164 # searcher1.search() # find first path
165 # searcher1.search() # find next path
166 # searcher2 = AStarSearcher(searchExample.simp_delivery_graph) # A*
167 # searcher2.search() # find first path
168 # searcher2.search() # find next path
169 # searcher3 = Searcher(searchExample.cyclic_simp_delivery_graph) # DFS
170 # searcher3.search() # find first path with DFS. What do you expect to
171 #   happen?
172 # searcher4 = AStarSearcher(searchExample.cyclic_simp_delivery_graph) # A*
173 # searcher4.search() # find first path
174
175 # To use the GUI for A* search do the following
176 # python -i searchGUI.py
177 # SearcherGUI(AStarSearcher, searchExample.simp_delivery_graph)
178 # 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 graphs with cycles, and graphs 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     for var in self.variables:
140         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

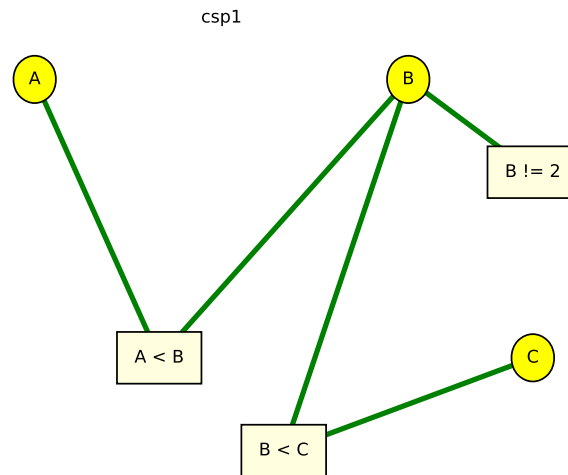
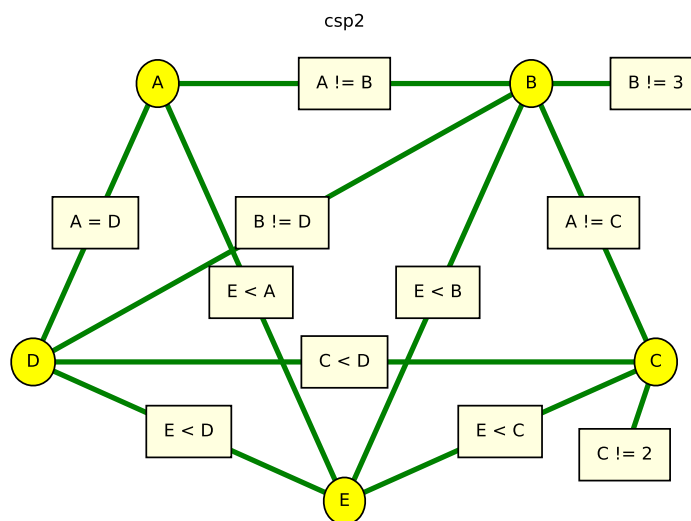
```

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()`

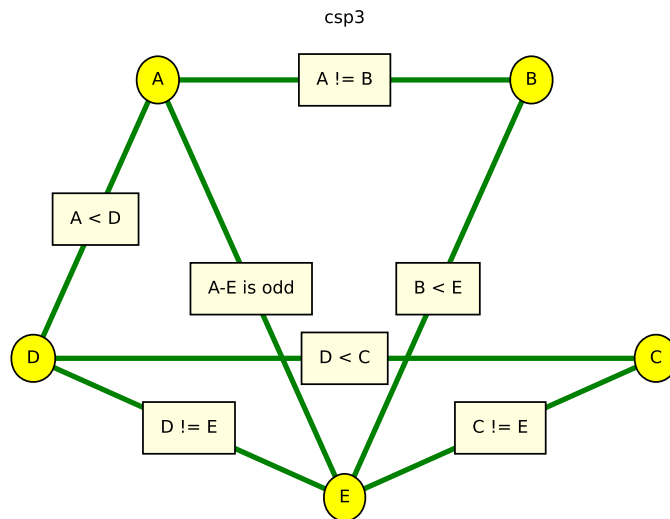


Figure 4.3: csp3.show()

```

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")])

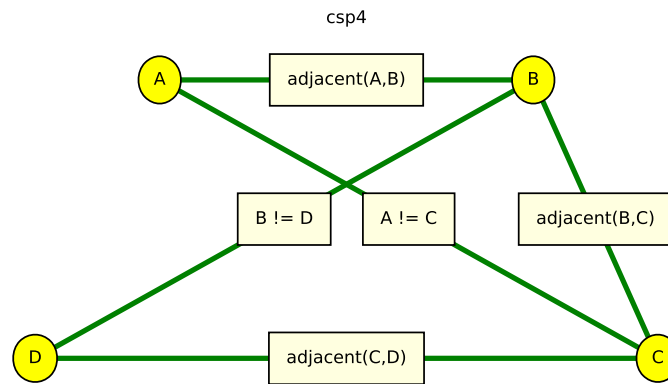
```

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?

```

_____cspExamples.py — (continued) _____
75 def adjacent(x,y):
76     """True when x and y are adjacent numbers"""
77     return abs(x-y) == 1
78
79 csp4 = CSP("csp4", {A,B,C,D},
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.

```

_____cspExamples.py — (continued) _____
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.

```

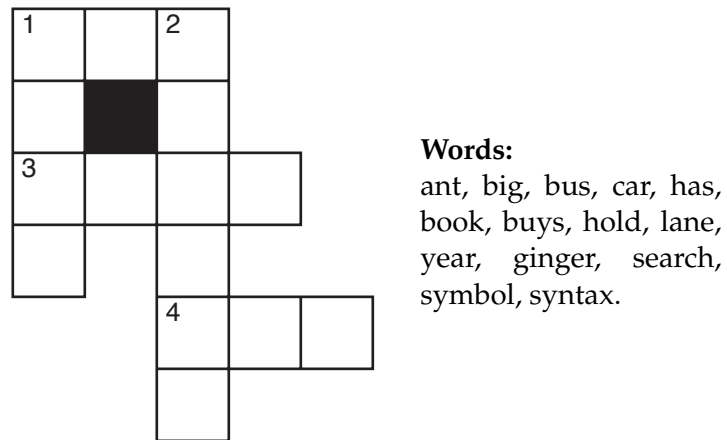


Figure 4.5: crossword1: a crossword puzzle to be solved

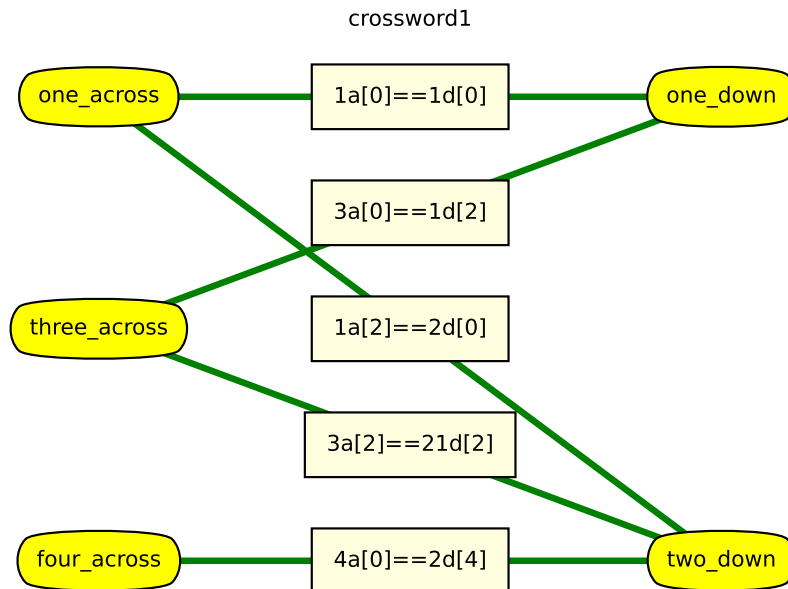


Figure 4.6: crossword1.show()

```

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

```



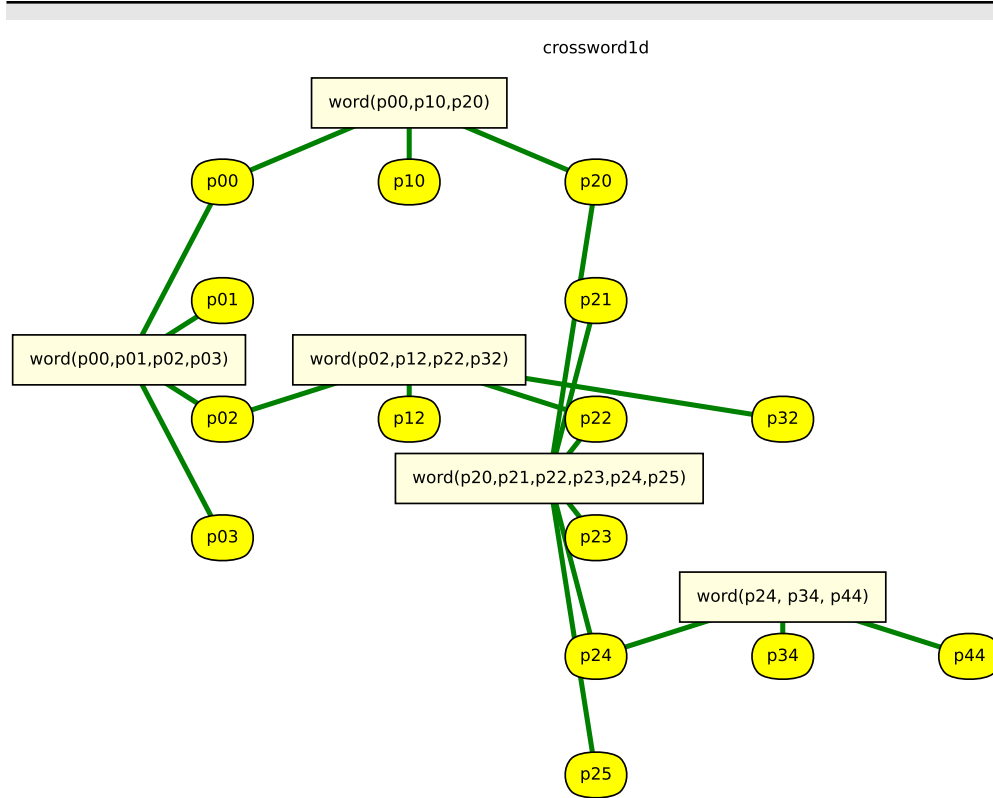


Figure 4.7: crossword1d.show()

```

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 ( $8 \times 8$ ) 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

```

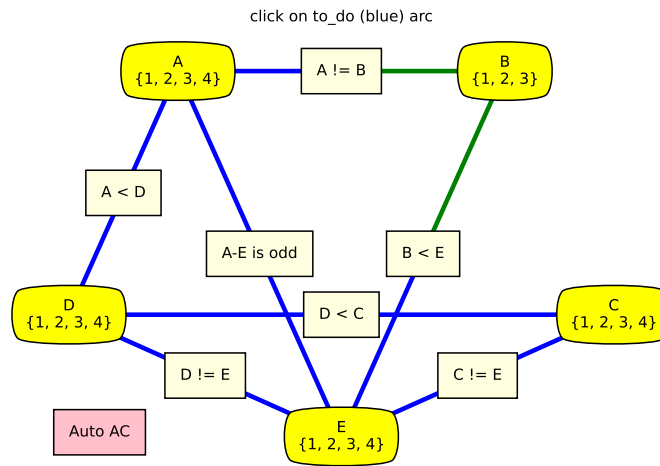


Figure 4.8: ConsistencyGUI(cspExamples.csp3).go()

**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         self.quitting = False
25         Con_solver.__init__(self, csp, **kwargs)
26         csp.show(showAutoAC = True)
27         csp.fig.canvas.mpl_connect('close_event', self.window_closed)
28
29     def go(self):
30         try:
31             res = self.solve_all()
32             self.csp.draw_graph(domains=self.domains,
33                               title="No more solutions. GUI finished. ",
34                               fontsize=self.fontsize)
35             return res
36         except ExitToPython:
37             print("GUI closed")
38
39     def select_arc(self, to_do):
40         while True:
41             self.csp.draw_graph(domains=self.domains, to_do=to_do,
42                               title="click on to_do (blue) arc",
43                               fontsize=self.fontsize)
44
45             self.wait_for_user()
46             if self.csp.autoAC:
47                 break
48             picked = self.csp.picked
49             self.csp.picked = None
50             if picked in to_do:

```

```

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

```

```

97         line.set_linewidth(self.csp.linewidth)
98         #elif n==4 and self.add_to_do: # adding to to_do
99         #     print("adding to to_do",self.add_to_do) ## highlight these arc
100
101     def wait_for_user(self):
102         while self.csp.picked == None and not self.csp.autoAC and not
            self.quitting:
103             plt.pause(0.01) # controls reaction time of GUI
104         if self.quitting:
105             raise ExitToPython()
106
107     def window_closed(self, event):
108         self.quitting = True
109
110 class ExitToPython(Exception):
111     pass
112
113 import cspExamples
114 # Try:
115 # ConsistencyGUI(cspExamples.csp1).go()
116 # ConsistencyGUI(cspExamples.csp3).go()
117 # ConsistencyGUI(cspExamples.csp3, speed=4, fontsize=15).go()
118
119 if __name__ == "__main__":
120     print("Try e.g.: ConsistencyGUI(cspExamples.csp3).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
49         conflict) is selected
49     prob_anycon is the probability that a variable in any conflict is
         selected

```

```

50         (otherwise a variable is chosen at random)
51         """
52         if self.current_assignment is None:
53             self.restart()
54             self.number_of_steps += 1
55             if not self.conflicts:
56                 self.display(1, "Solution found:", self.current_assignment,
57                             "after restart")
58                 return self.number_of_steps
59             if prob_best > 0: # we need to maintain a variable priority queue
60                 return self.search_with_var_pq(max_steps, prob_best,
61                                                 prob_anycon)
62         else:
63             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)

```

```

82         self.current_assignment[var]=val
83         for varcon in self.csp.var_to_const[var]:
84             if varcon.holds(self.current_assignment):
85                 if varcon in self.conflicts:
86                     self.conflicts.remove(varcon)
87             else:
88                 if varcon not in self.conflicts:
89                     self.conflicts.add(varcon)
90         self.display(2,"    Number of conflicts",len(self.conflicts))
91     if not self.conflicts:
92         self.display(1,"Solution found:", self.current_assignment,
93                     "in", self.number_of_steps,"steps")
94         return self.number_of_steps
95     self.display(1,"No solution in",self.number_of_steps,"steps",
96                 len(self.conflicts),"conflicts remain")
97     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
132                             conflicts
133                             var_differential[v] =
134                                 var_differential.get(v,0)-1
135                 else:
136                     if varcon not in self.conflicts: # was consis, not now
137                         self.display(3,"Became inconsistent",varcon)
138                         self.conflicts.add(varcon)
139                         for v in varcon.scope: # v is in one more
140                             conflicts
141                             var_differential[v] =
142                                 var_differential.get(v,0)+1
143             self.variable_pq.update_each_priority(var_differential)
144             self.display(2,"Number of conflicts",len(self.conflicts))
145         if not self.conflicts: # no conflicts, so solution found
146             self.display(1,"Solution found:",
147                 self.current_assignment,"in",
148                 self.number_of_steps,"steps")
149             return self.number_of_steps
150         self.display(1,"No solution in",self.number_of_steps,"steps",
151             len(self.conflicts),"conflicts remain")
152     return None

```

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

---

cspSLS.py — (continued)

---

```

149     def create_pq(self):
150         """Create the variable to number-of-conflicts priority queue.

```

```

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

```

---

cspSLS.py — (continued)

```

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

```

**Exercise 4.15** 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

```

```

    removing it.
    If the priority queue is empty, IndexError is raised.
    """
    self.max_size = max(self.max_size, len(self.pq)) # keep statistics
    triple = self.pq[0]
    while triple[2] == self.REMOVED:
        heapq.heappop(self.pq)
        triple = self.pq[0]
    return triple[2], triple[0] # elt, value

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

```

#### 4.5.4 Plotting Run-Time Distributions

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

```

cspSLS.py — (continued)
import matplotlib.pyplot as plt
# plt.style.use('grayscale')

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

    def plot_runs(self, num_runs=100, max_steps=1000, prob_best=1.0,
                  prob_anycon=1.0):
        """Plots num_runs of SLS for the given settings.
        """
        stats = []
        SLSearcher.max_display_level, temp_md1 = 0,
        SLSearcher.max_display_level # no display
        for i in range(num_runs):
            searcher = SLSearcher(self.csp)
            num_steps = searcher.search(max_steps, prob_best, prob_anycon)
            if num_steps:

```

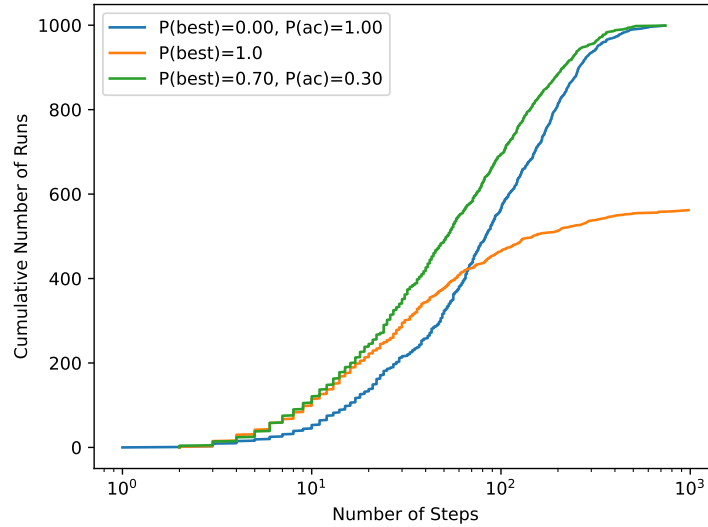


Figure 4.9: Run-time distributions for three algorithms on *csp2*.

```

262         stats.append(num_steps)
263     stats.sort()
264     if prob_best >= 1.0:
265         label = "P(best)=1.0"
266     else:
267         p_ac = min(prob_anycon, 1-prob_best)
268         label = "P(best)=%.2f, P(ac)=%.2f" % (prob_best, p_ac)
269     plt.plot(stats, range(len(stats)), label=label)
270     plt.legend(loc="upper left")
271     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.

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

```

```
103 | # bnb.max_display_level=3 # show more detail
104 | # 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:

---

```

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.

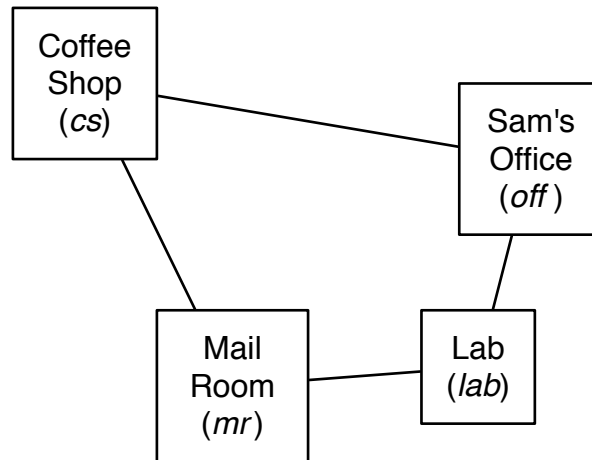
**Features to describe states***RLoc* – Rob's location*RHC* – Rob has coffee*SWC* – Sam wants coffee*MW* – Mail is waiting*RHM* – Rob has mail**Actions***mc* – move clockwise*mcc* – move counterclockwise*puc* – pickup coffee*dc* – deliver coffee*pum* – pickup mail*dm* – deliver mail

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)

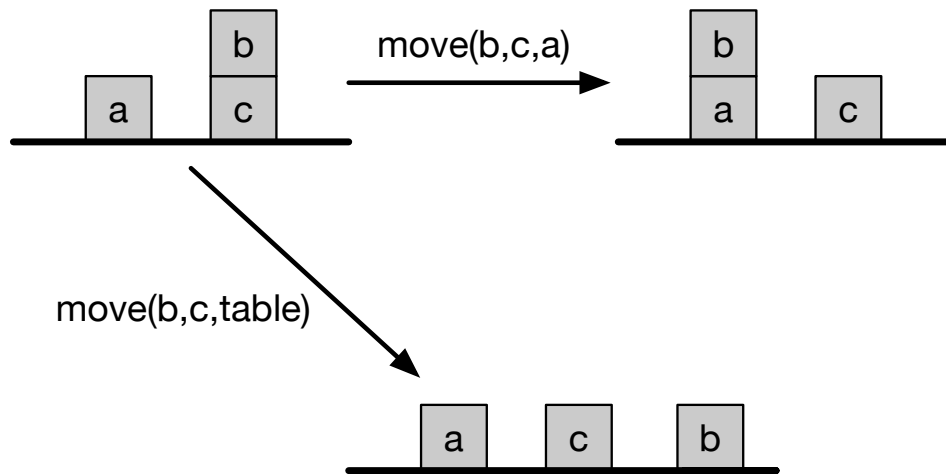


Figure 6.2: Blocks world with two actions

```

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'})

```

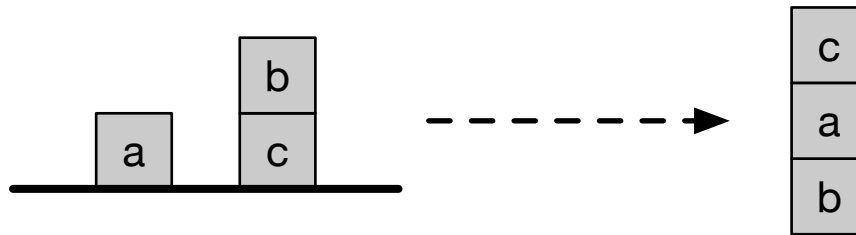


Figure 6.3: Blocks problem blocks1

```

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

**Exercise 6.9** After completing the previous exercise, design incompatible assignments for the blocks world. (This 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
16       action and time
17     * the dynamics are specified by the STRIPS representation of actions
18     """
19
20     def __init__(self, planning_problem, number_stages=2):
21         prob_domain = planning_problem.prob_domain
22         initial_state = planning_problem.initial_state
23         goal = planning_problem.goal
24         # 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
        for t in range(number_stages+1)}
67     CSP.__init__(self, "CSP_from_Strips", variables, constraints)
68
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
        action is not in {actset}"
95     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
                                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
                    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,
                                POP_node(new_actions,consts4,new_agenda1,new_cls),
117                                         cost=1)
118 
```

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.



`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, target_index=0, prob_test=0.10,
23                 prob_valid=0.11,
24                 header=None, target_type= None, one_hot=False,
25                 seed=None): #12345):
26         """A dataset for learning.
27         train is a list of tuples representing the training examples
28         test is the list of tuples representing the test examples
29         if test is None, a test set is created by selecting each
30         example with probability prob_test
31         target_index is the index of the target.
32         If negative, it counts from right.
33         If target_index is larger than the number of properties,
34         there is no target (for unsupervised learning)
35         prob_valid is the probability of putting a training example in a
36         validation set
37         header is a list of names for the features
38         target_type is either None for automatic detection of target type
39         or one of "numeric", "boolean", "categorical"
40         one_hot is True gives a one-hot encoding of categorical features
41         seed is for random number; None gives a different test set each time
42         """
43         if seed: # given seed makes partition consistent from run-to-run
44             random.seed(seed)
45         if test is None:
46             train,test = partition_data(train, prob_test)
47             self.train, self.valid = partition_data(train, prob_valid)
48             self.test = test
49
50         self.display(1,"Training set has",len(self.train),"examples. Number
51             of columns: ",{len(e) for e in self.train})
52         self.display(1,"Test set has",len(test),"examples. Number of
53             columns: ",{len(e) for e in test})
54         self.display(1,"Validation set has",len(self.valid),"examples.
55             Number of columns: ",{len(e) for e in self.valid})
56         self.prob_test = prob_test

```

```

51 self.num_properties = len(self.train[0])
52 if target_index < 0: #allows for -1, -2, etc.
53     self.target_index = self.num_properties + target_index
54 else:
55     self.target_index = target_index
56 self.header = header
57 self.domains = [set() for i in range(self.num_properties)]
58 for example in self.train:
59     for ind,val in enumerate(example):
60         self.domains[ind].add(val)
61 self.conditions_cache = {} # cache for computed conditions
62 self.create_features(one_hot)
63 if target_type:
64     self.target.ftype = target_type
65 self.display(1,"There are",len(self.input_features),"input
    features")
66
67 def __str__(self):
68     if self.train and len(self.train)>0:
69         return ("Data: "+str(len(self.train))+ " training examples, "
70             +str(len(self.test))+ " test examples, "
71             +str(len(self.train[0]))+" features.")
72     else:
73         return ("Data: "+str(len(self.train))+ " training examples, "
74             +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) —
76 def create_features(self, one_hot=False):
77     """create the set of features.
78     if one_hot==True then make categorical features into booleans for
79         each value
80     """
81     self.target = None
82     self.input_features = []
83     for i in range(self.num_properties):
84         frange = list(self.domains[i])
85         ftype = self.infer_type(frange)
86         if one_hot and ftype == "categorical" and i !=
            self.target_index:
87             for val in frange:
88                 def feat(e,index=i,val=val):
89                     return e[index]==val
90                 if self.header:
91                     feat.__doc__ = self.header[i]+"="+val
92                 else:
93                     feat.__doc__ = f"e[{i}]= {val}"

```



```

93         feat.frange = boolean
94         feat.type = "boolean"
95         self.input_features.append(feat)
96     else:
97         def feat(e, index=i):
98             return e[index]
99         if self.header:
100             feat.__doc__ = self.header[i]
101         else:
102             feat.__doc__ = "e["+str(i)+"]"
103         feat.frange = frange
104         feat.ftype = ftype
105         if i == self.target_index:
106             self.target = feat
107         else:
108             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)
110 def infer_type(self, domain):
111     """Infers the type of a feature with domain
112     """
113     if all(v in {True, False} for v in domain) or all(v in {0, 1} for v
114         in domain):
115         return "boolean"
116     if all(isinstance(v, (float, int)) for v in domain):
117         return "numeric"
118     else:
119         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.

```

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

```

```

156         return e[ind_] < cutat
157
158     if self.header:
159         feat.__doc__ = self.header[ind]+"<" +str(cutat)
160     else:
161         feat.__doc__ = "e["+str(ind)+"]<" +str(cutat)
162     feat.frange = boolean
163     feat.ftype = "boolean"
164     conds.append(feat)
165
166     else:
167         # create an indicator function for every value
168         for val in frange:
169             def feat(e, ind_=ind, val_=val):
170                 return e[ind_] == val_
171             if self.header:
172                 feat.__doc__ = self.header[ind]+"==" +str(val)
173             else:
174                 feat.__doc__ = "e["+str(ind)+"]=="+str(val)
175             feat.frange = boolean
176             feat.ftype = "boolean"
177             conds.append(feat)
178
179     self.conditions_cache[(max_num_cuts, categorical_only)] = conds
180     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).

```

180         _learnProblem.py — (continued)
181
182     def evaluate_dataset(self, data, predictor, error_measure):
183         """Evaluates predictor on data according to the error_measure
184         predictor is a function that takes an example and returns a
185         prediction for the target features.
186         error_measure(prediction,actual) -> non-negative real
187         """
188         if data:
189             try:
190                 value = statistics.mean(error_measure(predictor(e),
191                 self.target(e))
192                 for e in data)
193             except ValueError: # if error_measure gives an error
194                 return float("inf") # infinity
195             return value
196         else:
197             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.

```

196         _learnProblem.py — (continued)
197
198     class Evaluate(object):
199         """A container for the evaluation measures"""
200
201         def squared_loss(prediction, actual):
202             "squared loss "
203             if isinstance(prediction, (list,dict)):
204                 return (1-prediction[actual])**2 # the correct value is 1
205             else:
206                 return (prediction-actual)**2
207
208         def absolute_loss(prediction, actual):
209             "absolute loss "
210             if isinstance(prediction, (list,dict)):
211                 return abs(1-prediction[actual]) # the correct value is 1
212             else:
213                 return abs(prediction-actual)
214
215         def log_loss(prediction, actual):

```

```

214         "log loss (bits)"
215     try:
216         if isinstance(prediction, (list, dict)):
217             return -math.log2(prediction[actual])
218         else:
219             return -math.log2(prediction) if actual==1 else
                -math.log2(1-prediction)
220     except ValueError:
221         return float("inf") # infinity
222
223 def accuracy(prediction, actual):
224     "accuracy      "
225     if isinstance(prediction, dict):
226         prev_val = prediction[actual]
227         return 1 if all(prev_val >= v for v in prediction.values())
                else 0
228     if isinstance(prediction, list):
229         prev_val = prediction[actual]
230         return 1 if all(prev_val >= v for v in prediction) else 0
231     else:
232         return 1 if abs(actual-prediction) <= 0.5 else 0
233
234 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)
236 def partition_data(data, prob_test=0.30):
237     """partitions the data into a training set and a test set, where
238     prob_test is the probability of each example being in the test set.
239     """
240     train = []
241     test = []
242     for example in data:
243         if random.random() < prob_test:
244             test.append(example)
245         else:
246             train.append(example)
247     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)
249 class Data_from_file(Data_set):
250     def __init__(self, file_name, separator=',', num_train=None,
        prob_test=0.10, prob_valid=0.11,
251         has_header=False, target_index=0, one_hot=False,
252         categorical=[], target_type= None, include_only=None,
        seed=None): #seed=12345):
253         """create a dataset from a file
254         separator is the character that separates the attributes
255         num_train is a number specifying the first num_train tuples are
        training, or None
256         prob_test is the probability an example should in the test set (if
        num_train is None)
257         has_header is True if the first line of file is a header
258         target_index specifies which feature is the target
259         one_hot specifies whether categorical features should be encoded as
        one_hot.
260         categorical is a set (or list) of features that should be treated
        as categorical
261         target_type is either None for automatic detection of target type
262         or one of "numeric", "boolean", "categorical"
263         include_only is a list or set of indexes of columns to include
264         """
265         with open(file_name,'r',newline='') as csvfile:
266             self.display(1,"Loading",file_name)
267             # data_all = csv.reader(csvfile,delimiter=separator) # for more
        complicated CSV files
268             data_all = (line.strip().split(separator) for line in csvfile)
269             if include_only is not None:

```

```

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

```

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

```

learnProblem.py — (continued)
289 class Data_from_files(Data_set):
290     def __init__(self, train_file_name, test_file_name, separator=',',
291                 has_header=False, target_index=0, one_hot=False,
292                 categorical=[], target_type= None, include_only=None):
293         """create a dataset from separate training and file
294         separator is the character that separates the attributes
295         num_train is a number specifying the first num_train tuples are
            training, or None
296         prob_test is the probability an example should in the test set (if
            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
            one-hot
300         categorical is a set (or list) of features that should be treated
            as categorical
301         target_type is either None for automatic detection of target type
302             or one of "numeric", "boolean", "categorical"
303         include_only is a list or set of indexes of columns to include
304         """

```

```

305 with open(train_file_name,'r',newline='') as train_file:
306     with open(test_file_name,'r',newline='') as test_file:
307         # data_all = csv.reader(csvfile,delimiter=separator) # for more
            complicated CSV files
308         train_data = (line.strip().split(separator) for line in
            train_file)
309         test_data = (line.strip().split(separator) for line in
            test_file)
310         if include_only is not None:
311             train_data = ([v for (i,v) in enumerate(line) if i in
                include_only]
                for line in train_data)
312             test_data = ([v for (i,v) in enumerate(line) if i in
                include_only]
                for line in test_data)
313         if has_header: # this assumes the training file has a header
            and the test file doesn't
314             header = next(train_data)
315         else:
316             header = None
317         train_tuples = [interpret_elements(d) for d in train_data if
            len(d)>1]
318         test_tuples = [interpret_elements(d) for d in test_data if
            len(d)>1]
319         Data_set.__init__(self,train_tuples, test_tuples,
320                             target_index=target_index, header=header,
321                             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)
324 def interpret_elements(str_list):
325     """make the elements of string list str_list numeric if possible.
326     Otherwise remove initial and trailing spaces.
327     """
328     res = []
329     for e in str_list:
330         try:
331             res.append(int(e))
332         except ValueError:
333             try:
334                 res.append(float(e))
335             except ValueError:
336                 se = e.strip()
337                 if se in ["True","true","TRUE"]:
338                     res.append(True)
339                 elif se in ["False","false","FALSE"]:
340                     res.append(False)
341                 else:

```



```

342         res.append(e.strip())
343     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)
345 class Data_set_augmented(Data_set):
346     def __init__(self, dataset, unary_functions=[], binary_functions=[],
347                 include_orig=True):
348         """creates a dataset like dataset but with new features
349         unary_function is a list of unary feature constructors
350         binary_functions is a list of binary feature combiners.
351         include_orig specifies whether the original features should be
352             included
353         """
354         self.orig_dataset = dataset
355         self.unary_functions = unary_functions
356         self.binary_functions = binary_functions
357         self.include_orig = include_orig
358         self.target = dataset.target
359         Data_set.__init__(self, dataset.train, test=dataset.test,
360                           target_index = dataset.target_index)
361
362     def create_features(self, one_hot=False):
363         """create the set of features.
364         one_hot is ignored, but could be implemented as in
365             Data_set.create_features
366         """
367         if self.include_orig:
368             self.input_features = self.orig_dataset.input_features.copy()
369         else:
370             self.input_features = []
371         for u in self.unary_functions:
372             for f in self.orig_dataset.input_features:
373                 self.input_features.append(u(f))
374         for b in self.binary_functions:
375             for f1 in self.orig_dataset.input_features:
376                 for f2 in self.orig_dataset.input_features:

```

```

374         if f1 != f2:
375             self.input_features.append(b(f1,f2))

```

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

```

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

```

Example:

```

learnProblem.py — (continued)
420 # from learnProblem import Data_set_augmented, prod_feat
421 # data = Data_from_file('data/holiday.csv', has_header=True, num_train=19,
    target_index=-1)
422 # data = Data_from_file('data/iris.data', prob_test=1/3, target_index=-1)
423 ## Data = Data_from_file('data/SPECT.csv', prob_test=0.5, target_index=0)
424 # dataplus = Data_set_augmented(data, [], [prod_feat])
425 # 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)
426 from display import Displayable
427
428 class Learner(Displayable):
429     def __init__(self, dataset):
430         raise NotImplementedError("Learner.__init__") # abstract method
431
432     def learn(self):
433         """returns a predictor, a function from a tuple to a value for the
            target feature
434         """
435         raise NotImplementedError("learn") # abstract method
436
437     def predictor_string(self, sig_dig=3):
438         """String representation of the learned predictor
439         """
440         return "no representation"

```

## 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 of the values are ratings in  $\{1, 2, 3, 4, 5\}$ , we can only predict one of these integers.
- a point prediction, where we are allowed to predict any value. For example, if the values of the feature are  $\{0, 1\}$  we may be allowed to predict 0.3, 1, or even 1.7. For all of the criteria 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}

```

```

26         for e in data:
27             counts[e] += 1
28         s = sum(counts.values())
29         return {k:v/s for (k,v) in counts.items()}
30
31     def bounded_empirical(data, domain=[0,1], bound=0.01):
32         "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]):

```

```

74         "median"
75         return statistics.median(data)
76
77     all = [empirical, mean, rmean, bounded_empirical, laplace, cmode, mode,
78           median, cmedian]
79
80     # The following suggests appropriate predictions as a function of the
81     # target type
82     select = {"boolean": [empirical, bounded_empirical, laplace, cmode,
83                           cmedian],
84              "categorical": [empirical, bounded_empirical, laplace, cmode,
85                              cmedian],
86              "numeric": [mean, rmean, mode, median]}

```

### 7.3.1 Evaluation

To evaluate a point prediction, 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,
85                   num_samples=10000,
86                   test_size=10, training_sizes=
87                       [1,2,3,4,5,10,20,100,1000]):
88     for train_size in training_sizes:
89         results = {predictor: {error_measure: 0 for error_measure in
90                               error_measures}
91                   for predictor in Predict.all}
92         for sample in range(num_samples):
93             prob = random.random()
94             training = [1 if random.random()<prob else 0 for i in
95                         range(train_size)]
96             test = [1 if random.random()<prob else 0 for i in
97                     range(test_size)]
98             for predictor in Predict.all:
99                 prediction = predictor(training)
100                 for error_measure in error_measures:
101                     results[predictor][error_measure] += sum(
102                         error_measure(prediction, actual)
103                         for actual in
104                             test) /
105                             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
                           error_measures), sep="\t")
105
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
           each split
19         leaf_prediction=Predict.empirical, # what to use for value
           at leaves
20         train=None, # used for cross validation
21         max_num_cuts=8, # maximum number of conditions to split a
           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)
53             self.display(2,f"leaf prediction for {len(data_subset)}
54                         examples is {prediction}")
55             def leaf_fun(e):
56                 return prediction
57             leaf_fun.__doc__ = str(prediction)
58             leaf_fun.num_leaves = 1
59             return leaf_fun
60         else: # a split succeeded
61             false_examples, true_examples = partn
62             rem_features = [fe for fe in conditions if fe != split]
63             self.display(2,"Splitting on",split.__doc__,"with examples
64                         split",
65                         len(true_examples),":",len(false_examples))
66             true_tree = self.learn_tree(rem_features,true_examples)
67             false_tree = self.learn_tree(rem_features,false_examples)
68             def fun(e):
69                 if split(e):
70                     return true_tree(e)
71                 else:
72                     return false_tree(e)
73             #fun = lambda e: true_tree(e) if split(e) else false_tree(e)
74             fun.__doc__ = (f"(if {split.__doc__} then {true_tree.__doc__}"
75                          f" else {false_tree.__doc__})")
76             fun.num_leaves = true_tree.num_leaves + false_tree.num_leaves
77             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

```

```

91     best_error = self.sum_losses(data_subset) - self.gamma
92     self.display(3, " no split has
          error=", best_error, "with", len(conditions), "conditions")
93     best_partition = None
94     for feat in conditions:
95         false_examples, true_examples = partition(data_subset, feat)
96         if
97             min(len(false_examples), len(true_examples)) >= self.min_child_weight:
98                 err = (self.sum_losses(false_examples)
99                     + self.sum_losses(true_examples))
100                 self.display(3, " split on", feat.__doc__, "has error=", err,
          "splits
          into", len(true_examples), ":", len(false_examples), "gamma=", self.gamma)
101                 if err < best_error:
102                     best_feat = feat
103                     best_error = err
104                     best_partition = false_examples, true_examples
105     self.display(2, "best split is on", best_feat.__doc__,
106                 "with err=", best_error)
107     return best_feat, best_partition
108
109     def sum_losses(self, data_subset):
110         """returns sum of losses for dataset (with no more splits)
111         There a single prediction for all leaves using leaf_prediction
112         It is evaluated using split_to_optimize
113         """
114         prediction = self.leaf_value(data_subset, self.target.frange)
115         error = sum(self.split_to_optimize(prediction, self.target(e))
116                    for e in data_subset)
117         return error
118
119     def partition(data_subset, feature):
120         """partitions the data_subset by the feature"""
121         true_examples = []
122         false_examples = []
123         for example in data_subset:
124             if feature(example):
125                 true_examples.append(example)
126             else:
127                 false_examples.append(example)
128         return false_examples, true_examples

```

Test cases:

---

```

131 from learnProblem import Data_set, Data_from_file
132
133 def testDT(data, print_tree=True, selections = None, **tree_args):
134     """Prints errors and the trees for various evaluation criteria and ways
          to select leaves.
135     """

```

```

136     if selections == None: # use selections suitable for target type
137         selections = Predict.select[data.target.ctype]
138     evaluation_criteria = Evaluate.all_criteria
139     print("Split Choice", "Leaf Choice\t", "#leaves", '\t'.join(ecrit.__doc__
140                                     for ecrit in
                                     evaluation_criteria), sep="\t")
141     for crit in evaluation_criteria:
142         for leaf in selections:
143             tree = DT_learner(data, split_to_optimize=crit,
144                             leaf_prediction=leaf,
145                             **tree_args).learn()
146             print(crit.__doc__, leaf.__doc__, tree.num_leaves,
147                   "\t".join("{:.7f}".format(data.evaluate_dataset(data.test,
148                             tree, ecrit))
149                           for ecrit in evaluation_criteria), sep="\t")
150
151     if print_tree:
152         print(tree.__doc__)
153
154 #DT_learner.max_display_level = 4
155 if __name__ == "__main__":
156     # Choose one of the data files
157     #data=Data_from_file('data/SPECT.csv', target_index=0);
158     print("SPECT.csv")
159     #data=Data_from_file('data/iris.data', target_index=-1);
160     print("iris.data")
161     data = Data_from_file('data/carbool.csv', target_index=-1, seed=123)
162     #data = Data_from_file('data/mail_reading.csv', target_index=-1);
163     print("mail_reading.csv")
164     #data = Data_from_file('data/holiday.csv', has_header=True,
165                           num_train=19, target_index=-1); print("holiday.csv")
166     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()
60     plt.xscale(xscale) # change between log and linear scale
61     plt.xlabel("min_child_weight")
62     plt.ylabel("average "+criterion.__doc__)
63     folded_data = K_fold_dataset(data, num_folds)
64     if maxx == None:
65         maxx = len(data.train)//2+1
66     errors = [] # validation errors
67     terrors = [] # test set errors
68     for mcw in range(1,maxx):
69         errors.append(folded_data.validation_error(DT_learner,criterion,leaf_prediction=leaf_prediction,
70                                                    min_child_weight=mcw))
71         tree = DT_learner(data, criterion, leaf_prediction=leaf_prediction,
72                           min_child_weight=mcw).learn()
73         terrors.append(data.evaluate_dataset(data.test,tree,criterion))
74     plt.plot(range(1,maxx), errors, ls='-',color='k',
75             label="validation for "+criterion.__doc__)
76     plt.plot(range(1,maxx), terrors, ls='--',color='k',
77             label="test set for "+criterion.__doc__)
78     plt.legend()
79     plt.draw()
80
81 # The following produces the graphs of Figure 7.18 of Poole and Mackworth
82 # [2023]
83 # data = Data_from_file('data/SPECT.csv',target_index=0, seed=123)
84 # plot_error(data, criterion=Evaluate.log_loss,
85 #             leaf_prediction=Predict.laplace)
86
87 #also try:
88 # plot_error(data)
89 # 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

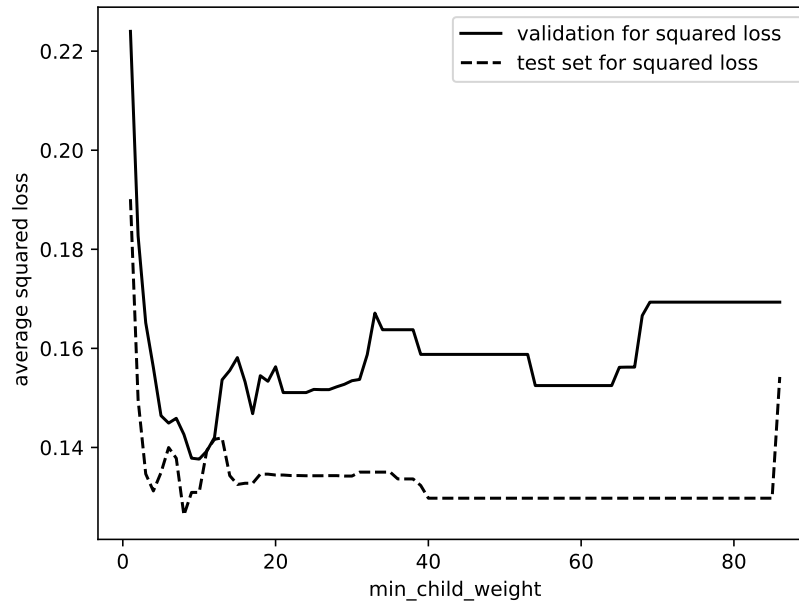


Figure 7.2: plot\_error for SPECT dataset

of the exercise of Section 7.6. Which criteria makes the most difference?

## 7.6 Linear Regression and Classification

Here is a stochastic gradient descent searcher for linear regression and classification.

```

learnLinear.py — Linear Regression and Classification
11 from learnProblem import Learner
12 import random, math
13
14 class Linear_learner(Learner):
15     def __init__(self, dataset, train=None,
16                 learning_rate=0.1, max_init = 0.2,
17                 squashed=True, batch_size=10):
18         """Creates a gradient descent searcher for a linear classifier.
19         The main learning is carried out by learn()
20
21         dataset provides the target and the input features
22         train provides a subset of the training data to use
23         number_iterations is the default number of steps of gradient descent
24         learning_rate is the gradient descent step size
25         max_init is the maximum absolute value of the initial weights

```

```

26     squashed specifies whether the output is a squashed linear function
27     """
28     self.dataset = dataset
29     self.target = dataset.target
30     if train==None:
31         self.train = self.dataset.train
32     else:
33         self.train = train
34     self.learning_rate = learning_rate
35     self.squashed = squashed
36     self.batch_size = batch_size
37     self.input_features = [one]+dataset.input_features # one is defined
38         below
39     self.weights = {feat:random.uniform(-max_init,max_init)
40                     for feat in self.input_features}

```

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

---

```

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

```

learn is the main algorithm of the learner. It does num\_iter steps of stochastic gradient descent. Only the number of iterations is specified; the other parameters it gets from the class.

---

```

learnLinear.py — (continued)
60     def learn(self, num_iter=100):
61         batch_size = min(self.batch_size, len(self.train))
62         d = {feat:0 for feat in self.weights}
63         for it in range(num_iter):
64             self.display(2,"prediction=",self.predictor_string())
65             for e in random.sample(self.train, batch_size):
66                 error = self.predictor(e) - self.target(e)

```



```

67         update = self.learning_rate*error
68         for feat in self.weights:
69             d[feat] += update*feat(e)
70         for feat in self.weights:
71             self.weights[feat] -= d[feat]
72             d[feat]=0
73         return self.predictor

```

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

```

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

```

*sigmoid*( $x$ ) is the function

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

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

```

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

```

*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,exps)}
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 validation 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)

```

```

142     #data = Data_from_file('data/SPECT.csv', target_index=0)
143     # data = Data_from_file('data/mail_reading.csv', target_index=-1)
144     # data = Data_from_file('data/carbool.csv', target_index=-1)
145     #random.seed(None) # reset seed
146     if learner is None:
147         learner = Linear_learner(data)
148     train_errors = []
149     valid_errors = []
150     for i in range(1,num_steps+1,step):
151         valid_errors.append(data.evaluate_dataset(data.valid,
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             "Valid error:",valid_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),valid_errors,ls='--',label=legend_label+"valid")
160     plt.legend()
161     plt.draw()
162     learner.display(1, "Train error:",train_errors[-1],
163         "Valid error:",valid_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_valid=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 unstable?] 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) —

---

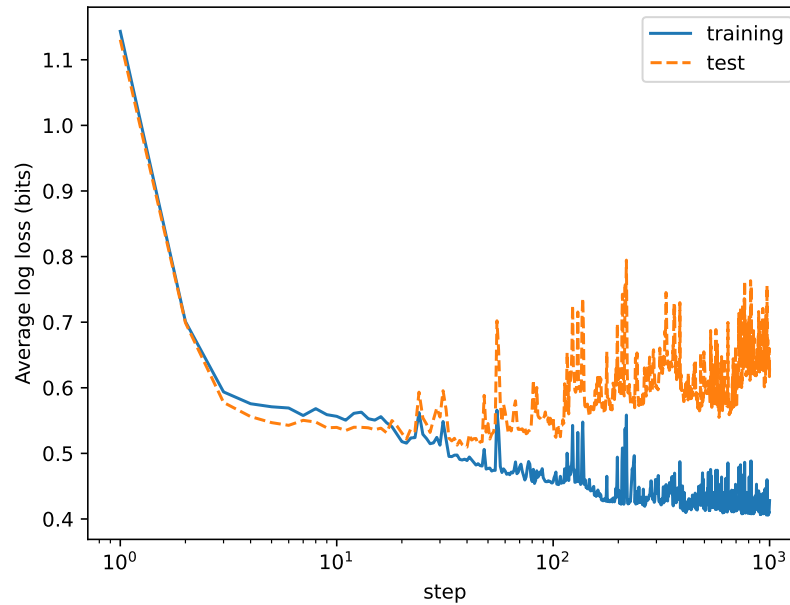


Figure 7.3: plot\_steps for SPECT dataset

```

173 def arange(start,stop,step):
174     """returns enumeration of values in the range [start,stop) separated by
175         step.
176         like the built-in range(start,stop,step) but allows for integers and
177         floats.
178         Note that rounding errors are expected with real numbers. (or use
179         numpy.arange)
180     """
181     while start<stop:
182         yield start
183         start += step
184
185 def plot_prediction(data,
186                     learner = None,
187                     minx = 0,
188                     maxx = 5,
189                     step_size = 0.01, # for plotting
190                     label = "function"):
191     plt.ion()
192     plt.xlabel("x")
193     plt.ylabel("y")
194     if learner is None:
195         learner = Linear_learner(data, squashed=False)
196     learner.learning_rate=0.001

```

```

194 learner.learn(100)
195 learner.learning_rate=0.0001
196 learner.learn(1000)
197 learner.learning_rate=0.00001
198 learner.learn(10000)
199 learner.display(1,"function learned is", learner.predictor_string(),
200                 "error=",data.evaluate_dataset(data.train, learner.predictor,
201                 Evaluate.squared_loss))
201 plt.plot([e[0] for e in data.train],[e[-1] for e in
202         data.train],"bo",label="data")
202 plt.plot(list(arange(minx,maxx,step_size)),[learner.predictor([x])
203         for x in
204         arange(minx,maxx,step_size)],
205         label=label)
204 plt.legend()
205 plt.draw()
206

```

---

learnLinear.py — (continued)

---

```

208 from learnProblem import Data_set_augmented, power_feat
209 def plot_polynomials(data,
210                     learner_class = Linear_learner,
211                     max_degree = 5,
212                     minx = 0,
213                     maxx = 5,
214                     num_iter = 1000000,
215                     learning_rate = 0.00001,
216                     step_size = 0.01, # for plotting
217                     ):
218     plt.ion()
219     plt.xlabel("x")
220     plt.ylabel("y")
221     plt.plot([e[0] for e in data.train],[e[-1] for e in
222             data.train],"ko",label="data")
222     x_values = list(arange(minx,maxx,step_size))
223     line_styles = ['-','--','-.-',':']
224     colors = ['0.5','k','k','k','k']
225     for degree in range(max_degree):
226         data_aug = Data_set_augmented(data,[power_feat(n) for n in
227             range(1,degree+1)],
228                                     include_orig=False)
228         learner = learner_class(data_aug,squashed=False)
229         learner.learning_rate = learning_rate
230         learner.learn(num_iter)
231         learner.display(1,"For degree",degree,
232                         "function learned is", learner.predictor_string(),
233                         "error=",data.evaluate_dataset(data.train,
234                                                         learner.predictor, Evaluate.squared_loss))
234         ls = line_styles[degree % len(line_styles)]
235         col = colors[degree % len(colors)]
236         plt.plot(x_values,[learner.predictor([x]) for x in x_values],

```

```

237         linestyle=ls, color=col,
                label="degree="+str(degree))
238     plt.legend(loc='upper left')
239     plt.draw()
240
241 # Try:
242 # data0 = Data_from_file('data/simp_regr.csv', prob_test=0, prob_valid=0,
                one_hot=False, target_index=-1)
243 # plot_prediction(data0)
244 # plot_polynomials(data0)
245 # What if the step size was bigger?
246 # datam = Data_from_file('data/mail_reading.csv', target_index=-1)
247 # 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 =
                random.sample(base_dataset.train, int(subsample*len(base_dataset.train)))
25         self.test = base_dataset.test
26         #Data_set.__init__(self, base_dataset.train, base_dataset.test,
27         #                  base_dataset.prob_test, base_dataset.target_index)
28
29         #def create_features(self):
30         """creates new features - called at end of Data_set.init()
31         defines a new target

```

```

32     """
33     self.input_features = self.base_dataset.input_features
34     def newout(e):
35         return self.base_dataset.target(e) - self.offset_fun(e)
36     newout.frange = self.base_dataset.target.frange
37     newout.ftype = self.infer_type(newout.frange)
38     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 mean 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     plt.ion()

```



```

110 plt.xscale('linear') # change between log and linear scale
111 plt.xlabel("number of trees")
112 plt.ylabel("mean squared loss")
113 markers = (m+c for c in ['k','g','r','b','m','c','y'] for m in
114             ['-','--','-.',':'])
115 for (mcw,gamma,learner) in learners:
116     data.display(1,f"min_child_weight={mcw}, gamma={gamma}")
117     learner.learn(steps)
118     plt.plot(range(steps+1), learner.errors, next(markers),
119             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)

```

```

136         self.trees.append(tree)
137         self.display(1,f"""Iteration {i} treesize = {tree.num_leaves}
            train logloss={
138             self.dataset.evaluate_dataset(self.dataset.train,
                self.gtb_predictor, Evaluate.log_loss)
139             } test logloss={
140             self.dataset.evaluate_dataset(self.dataset.test,
                self.gtb_predictor, Evaluate.log_loss)}""")
141     return self.gtb_predictor
142
143     def gtb_predictor(self, example, extra=0):
144         """prediction for example,
145         extras is an extra contribution for this example being considered
146         """
147         return sigmoid(sum(t(example) for t in self.trees)+extra)
148
149     def leaf_value(self, egs, domain=[0,1]):
150         """value at the leaves for examples egs
151         domain argument is ignored"""
152         predActs = [(self.gtb_predictor(e),self.target(e)) for e in egs]
153         return sum(a-p for (p,a) in predActs) / (sum(p*(1-p) for (p,a) in
            predActs)+self.lambda_reg)
154
155
156     def sum_losses(self, data_subset):
157         """returns sum of losses for dataset (assuming a leaf is formed
            with no more splits)
158         """
159         leaf_val = self.leaf_value(data_subset)
160         error = sum(Evaluate.log_loss(self.gtb_predictor(e,leaf_val),
            self.target(e))
            for e in data_subset) + self.gamma
161
162     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 method to compute the output values from the inputs, a method to back-propagate the error, and a method update its parameters (if it has any) for a batch.

```
learnNN.py — Neural Network Learning
11 from display import Displayable
12 from learnProblem import Learner, Data_set, Data_from_file,
   Data_from_files, Evaluate
13 from learnLinear import sigmoid, one, softmax, indicator
14 import random, math, time
15
```

```

16 class Layer(Displayable):
17     def __init__(self, nn, num_outputs=None):
18         """Abstract layer class, must be overridden.
19         nn is the neural network this layer is part of
20         num_outputs is the number of outputs for this layer.
21         """
22         self.nn = nn
23         self.num_inputs = nn.num_outputs # nn output is layer's input
24         if num_outputs:
25             self.num_outputs = num_outputs
26         else:
27             self.num_outputs = self.num_inputs # same as the inputs
28         self.outputs= [0]*self.num_outputs
29         self.input_errors = [0]*self.num_inputs
30         self.weights = []
31
32     def output_values(self, input_values, training=False):
33         """Return the outputs for this layer for the given input values.
34         input_values is a list (of length self.num_inputs) of the inputs
35         returns a list of length self.num_outputs.
36         It can act differently when training and when predicting.
37         """
38         raise NotImplementedError("output_values") # abstract method
39
40     def backprop(self, out_errors):
41         """Backpropagate the errors on the outputs
42         errors is a list of output errors (of length self.num_outputs).
43         Returns list of input errors (of length self.num_inputs).
44
45         This is only called after corresponding output_values(),
46         which should remember relevant information
47         """
48         raise NotImplementedError("backprop") # abstract method
49
50 class Optimizer(Displayable):
51     def update(self, layer):
52         """updates parameters after a batch.
53         """
54         pass

```

### 8.1.1 Linear Layer

A linear layer maintains an array of weights. `self.weights[i][o]` is the weight between input `i` and output `o`. The bias is treated implicitly as the last input, so the weight of the bias for output `o` is `self.weights[self.num_inputs][o]`.

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  $[-limit, limit]$ . As in Keras, AIPython treats the bias separately; it is initialized to zero.

learnNN.py — (continued)

```

56 class Linear_complete_layer(Layer):
57     """a completely connected layer"""
58     def __init__(self, nn, num_outputs, limit=None):
59         """A completely connected linear layer.
60         nn is a neural network that the inputs come from
61         num_outputs is the number of outputs
62         the random initialization of parameters is in range [-limit,limit]
63         """
64         Layer.__init__(self, nn, num_outputs)
65         if limit is None:
66             limit = math.sqrt(6/(self.num_inputs+self.num_outputs))
67         # self.weights[i][o] is the weight between input i and output o
68         self.weights = [[random.uniform(-limit, limit)
69                         if i < self.num_inputs else 0
70                         for o in range(self.num_outputs)]
71                         for i in range(self.num_inputs+1)]
72         # self.weights[i][o] is the accumulated change for a batch.
73         self.delta = [[0 for o in range(self.num_outputs)]
74                       for i in range(self.num_inputs+1)]
75
76     def output_values(self, inputs, training=False):
77         """Returns the outputs for the input values.
78         It remembers the values for the backprop.
79         """
80         self.display(3, f"Linear layer inputs: {inputs}")
81         self.inputs = inputs
82         for out in range(self.num_outputs):
83             self.outputs[out] = (sum(self.weights[inp][out]*self.inputs[inp]
84                                   for inp in range(self.num_inputs))
85                               + self.weights[self.num_inputs][out])
86         self.display(3, f"Linear layer inputs: {inputs}")
87         return self.outputs
88
89     def backprop(self, errors):
90         """Backpropagate errors, update weights, return input error.
91         errors is a list of size self.num_outputs
92         Returns errors for layer's inputs of size
93         """
94         self.display(3, f"Linear Backprop. input: {self.inputs} output
95                       errors: {errors}")
96         for out in range(self.num_outputs):
97             for inp in range(self.num_inputs):
98                 self.input_errors[inp] = self.weights[inp][out] * errors[out]
99                 self.delta[inp][out] += self.inputs[inp] * errors[out]
100                 self.delta[self.num_inputs][out] += errors[out]
101         self.display(3, f"Linear layer backprop input errors:
102                       {self.input_errors}")
103         return self.input_errors

```

### 8.1.2 Stochastic Gradient Descent

The optimizers update the weights of a layer. In layers without weights, the weights list is empty. The optimizers save the relevant parameters in the layer. This always includes the weights and the gradient for the most recent batch (in `layer.delta`). An optimizer must zero `layer.delta` so the new batch can start anew.

```

_____learnNN.py — (continued) _____
103 class SGD(Optimizer):
104     """Vanilla SGD"""
105     def __init__(self, layer, parms={'lr':0.001}):
106         """layer is a layer, which contains weight and gradient matrices
107         Layers without weights have weights=[]
108         """
109         self.lr = parms['lr'] if 'lr' in parms else 0.001
110
111     def update(self, layer):
112         """update weights of layer after a batch.
113         """
114         for inp in range(len(layer.weights)):
115             for out in range(len(layer.weights[0])):
116                 layer.weights[inp][out] -= self.lr*layer.delta[inp][out]
117                 layer.delta[inp][out] = 0

```

### 8.1.3 ReLU Layer

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

```

_____learnNN.py — (continued) _____
119 class ReLU_layer(Layer):
120     """Rectified linear unit (ReLU)  $f(z) = \max(0, z)$ .
121     The number of outputs is equal to the number of inputs.
122     """
123     def __init__(self, nn):
124         Layer.__init__(self, nn)
125
126
127     def output_values(self, input_values, training=False):
128         """Returns the outputs for the input values.
129         It remembers the input values for the backprop.
130         """
131         self.input_values = input_values
132         for i in range(self.num_inputs):
133             self.outputs[i] = max(0, input_values[i])
134         return self.outputs
135
136     def backprop(self, out_errors):
137         """Returns the derivative of the errors"""
138         for i in range(self.num_inputs):

```

```

139         self.input_errors[i] = out_errors[i] if self.input_values[i]>0
           else 0
140     return self.input_errors

```

### 8.1.4 Sigmoid Layer

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

```

learnNN.py — (continued)
142 class Sigmoid_layer(Layer):
143     """sigmoids of the inputs.
144     The number of outputs is equal to the number of inputs.
145     Each output is the sigmoid of its corresponding input.
146     """
147     def __init__(self, nn):
148         Layer.__init__(self, nn)
149
150     def output_values(self, input_values, training=False):
151         """Returns the outputs for the input values.
152         It remembers the output values for the backprop.
153         """
154         for i in range(self.num_inputs):
155             self.outputs[i] = sigmoid(out_errors[i])
156         return self.outputs
157
158     def backprop(self, errors):
159         """Returns the derivative of the errors"""
160         for i in range(self.num_inputs):
161             self.input_errors[i] =
162                 input_values[i]*out_errors[i]*(1-out_errors[i])
163         return self.input_errors

```

## 8.2 Feedforward Networks

```

learnNN.py — (continued)
164 class NN(Learner):
165     def __init__(self, dataset, optimizer=SGD, parms={}):
166         """Creates a neural network for a dataset,
167         layers is the list of layers
168         """
169         self.dataset = dataset
170         self.output_type = dataset.target.fdtype
171         self.input_features = dataset.input_features
172         self.num_outputs = len(self.input_features) # empty NN
173         self.optimizer = optimizer
174         self.parms = parms

```

```

175     self.layers = []
176     self.bn = 0 # number of batches run
177     self.printed_heading = False
178
179     def add_layer(self, layer):
180         """add a layer to the network.
181         Each layer gets number of inputs from the previous layers outputs.
182         """
183         self.layers.append(layer)
184         #if hasattr(layer, 'weights'):
185         layer.optimizer = self.optimizer(layer, self.parms)
186         self.num_outputs = layer.num_outputs
187
188     def predictor(self, ex):
189         """Predicts the value of the first output for example ex.
190         """
191         values = [f(ex) for f in self.input_features]
192         for layer in self.layers:
193             values = layer.output_values(values)
194         return sigmoid(values[0]) if self.output_type == "boolean" \
195             else softmax(values, self.dataset.target.frange) if
196             self.output_type == "categorical" \
197             else values[0]

```

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

```

learnNN.py — (continued)
198     def learn(self, epochs=5, batch_size=32, num_iter = None,
199               report_each=10):
200         """Learns parameters for a neural network using stochastic gradient
201         decent.
202         epochs is the number of times through the data (on average)
203         batch_size is the maximum size of each batch
204         num_iter is the number of iterations over the batches
205         - overrides epochs if provided
206         report_each means print errors after each multiple of that number
207         of batches
208         """
209         self.batch_size = min(batch_size, len(self.dataset.train)) # don't
210         have batches bigger than training size
211         if num_iter is None:
212             num_iter = (epochs * len(self.dataset.train)) //
213             self.batch_size
214         self.report_each = report_each
215         #self.display(0, "Batch\t", "\t".join(criterion.__doc__ for criterion
216         in Evaluate.all_criteria))
217         for i in range(num_iter):
218             batch = random.sample(self.dataset.train, self.batch_size)
219             for e in batch:
220                 # compute all outputs
221                 values = [f(e) for f in self.input_features]

```



```

216         for layer in self.layers:
217             values = layer.output_values(values, training=True)
218             # backpropagate
219             predicted = [sigmoid(v) for v in values] if self.output_type
                == "boolean"\
220                 else softmax(values) if self.output_type ==
                    "categorical"\
221                 else values
222             actuals = indicator(self.dataset.target(e),
                self.dataset.target.frange) \
223                 if self.output_type == "categorical"\
224                 else [self.dataset.target(e)]
225             errors = [pred-obsd for (obsd,pred) in
                zip(actuals,predicted)]
226             for layer in reversed(self.layers):
227                 errors = layer.backprop(errors)
228             # Update all parameters in batch
229             for layer in self.layers:
230                 layer.optimizer.update(layer)
231             self.bn+=1
232             if (i+1)%report_each==0:
233                 self.trace(i)
234
235     def trace(self,i):
236         """print tracing of the batch updates"""
237         if not self.printed_heading:
238             self.display(0, "Errors on","validation" if self.dataset.valid
                else "training")
239             self.display(0,"batch\t", "\t".join(crit.__doc__ for crit in
                Evaluate.all_criteria))
240             self.printed_heading = True
241             self.display(0,self.bn,"\t",
242                 "\t\t".join("{:.4f}".format(
243                     self.dataset.evaluate_dataset(
244                         self.dataset.valid if self.dataset.valid else
                            self.dataset.train,
245                             self.predictor, criterion))
                for criterion in Evaluate.all_criteria), sep="")
246

```

## 8.3 Alternative Optimizers

The following are alternatives to stochastic gradient descent (SGD), defined in Section 8.1.2.

### 8.3.1 Momentum

learnNN.py — (continued)

```

248 class Momentum(Optimizer):
249     """SGD with momentum"""

```

```

250
251 """a completely connected layer"""
252 def __init__(self, layer, parms={'lr':0.01, 'momentum':0.9}):
253     """
254     lr is the learning rate
255     momentum is the momentum parameter of PyTorch or Keras
256
257     """
258     self.lr = parms['lr'] if 'lr' in parms else 0.01
259     self.momentum = parms['momentum'] if 'momentum' in parms else 0.9
260     layer.velocity = [[0 for _ in range((len(layer.weights[0])))]
261                       for _ in range(len(layer.weights))]
262
263
264 def update(self, layer):
265     """updates parameters after a batch with momentum"""
266     for inp in range(len(layer.weights)):
267         for out in range(len(layer.weights[0])):
268             layer.velocity[inp][out] =
269                 self.momentum*layer.velocity[inp][out] -
270                 self.lr*layer.delta[inp][out]
269             layer.weights[inp][out] += layer.velocity[inp][out]
270             layer.delta[inp][out] = 0

```

### 8.3.2 RMS-Prop

```

learnNN.py — (continued)
272 class RMS_Prop(Optimizer):
273     """a completely connected layer"""
274     def __init__(self, layer, parms={'rho':0.9, 'epsilon':1e-07,
275                                     'lr':0.01}):
276         """A completely connected linear layer.
277         nn is a neural network that the inputs come from
278         num_outputs is the number of outputs
279         max_init is the maximum value for random initialization of
280         parameters
281
282         """
283         # layer.ms[i][o] is running average of squared gradient input i and
284         # output o
285         layer.ms = [[0 for _ in range(len(layer.weights[0]))]
286                   for _ in range(len(layer.weights))]
287         self.rho = parms['rho'] if 'rho' in parms else 0.9
288         self.epsilon = parms['epsilon'] if 'epsilon' in parms else 1e-07
289         self.lr = parms['lr'] if 'lr' in parms else 0.01
290
291     def update(self, layer):
292         """updates parameters after a batch"""
293         for inp in range(len(layer.weights)):
294             for out in range(len(layer.weights[0])):

```

```

291         layer.ms[inp][out] = self.rho*layer.ms[inp][out]+
           (1-self.rho) * layer.delta[inp][out]**2
292         layer.weights[inp][out] -= self.lr * layer.delta[inp][out] /
           (layer.ms[inp][out]+self.epsilon)**0.5
293         layer.delta[inp][out] = 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?

**Exercise 8.2** Both Goodfellow, Bengio, and Courville [2016] and Poole and Mackworth [2023] find the gradient by dividing `self.delta[inp][out]` by the batch size, but some of the above code doesn't. To make code with dividing and without dividing the same, the step sizes need to be different by a factor of the batch size. Find a reasonable step size using an informal hyperparameter tuning; try some orders of magnitude of the step size to see what works best. What happens if the batch size is changed, but the step size is unchanged? (Try orders of magnitude difference in step sizes.) For each of the update methods, which works better: dividing by the step size or not?

## 8.4 Dropout

Dropout is implemented as a layer.

```

learnNN.py — (continued)
295 from utilities import flip
296 class Dropout_layer(Layer):
297     """Dropout layer
298     """
299
300     def __init__(self, nn, rate=0):
301         """
302         rate is fraction of the input units to drop. 0 <= rate < 1
303         """
304         self.rate = rate
305         Layer.__init__(self, nn)
306         self.mask = [0]*self.num_inputs
307
308
309     def output_values(self, input_values, training=False):
310         """Returns the outputs for the input values.
311         It remembers the input values and mask for the backprop.
312         """
313         if training:
314             scaling = 1/(1-self.rate)
315             for i in range(self.num_inputs):
316                 self.mask[i] = 0 if flip(self.rate) else 1
317                 input_values[i] = self.mask[i]*input_values[i]*scaling

```

```

318         return input_values
319
320     def backprop(self, output_errors):
321         """Returns the derivative of the errors"""
322         for i in range(self.num_inputs):
323             self.input_errors[i] = output_errors[i]*self.mask[i]
324         return self.input_errors

```

## 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)
326 #data = Data_from_file('data/mail_reading.csv', target_index=-1)
327 #data = Data_from_file('data/mail_reading_consis.csv', target_index=-1)
328 data = Data_from_file('data/SPECT.csv', target_index=0, seed=12345)
329 #data = Data_from_file('data/iris.data', target_index=-1)
330 #data = Data_from_file('data/if_x_then_y_else_z.csv', num_train=8,
331                       target_index=-1) # not linearly sep
332 #data = Data_from_file('data/holiday.csv', target_index=-1) #,
333                       num_train=19)
334 #data = Data_from_file('data/processed.cleveland.data', target_index=-1)
335 #random.seed(None)
336
337 # nn3 is has a single hidden layer of width 3
338 nn3 = NN(data, optimizer=SGD)
339 nn3.add_layer(Linear_complete_layer(nn3,3))
340 #nn3.add_layer(Sigmoid_layer(nn3))
341 nn3.add_layer(ReLU_layer(nn3))
342 nn3.add_layer(Linear_complete_layer(nn3,1)) # when using
343 output_type="boolean"
344 #
345 #
346 # Print some training examples
347 #for eg in random.sample(data.train,10): print(eg,nn3.predictor(eg))
348
349 # Print some test examples
350 #for eg in random.sample(data.test,10): print(eg,nn3.predictor(eg))
351
352 # To see the weights learned in linear layers
353 # nn3.layers[0].weights
354 # nn3.layers[2].weights
355
356 # nn3do is like nn3 but with dropout on the hidden layer
357 nn3do = NN(data, optimizer=SGD)
358 nn3do.add_layer(Linear_complete_layer(nn3do,3))

```

```

356 | nn3.add_layer(Sigmoid_layer(nn3)) # comment this or the next
357 | nn3do.add_layer(ReLU_layer(nn3do))
358 | nn3do.add_layer(Dropout_layer(nn3do, rate=0.5))
359 | nn3do.add_layer(Linear_complete_layer(nn3do,1))
360 | nn3do.learn(epochs=None, batch_size=100, num_iter = 1000, report_each=100)

```

`create_nn(dataset, architecture, optimizer, parameters)` creates a generic feedforward neural network. The dataset is needed as the input and output is determined by the data. The architecture is a list of the sizes of hidden layers. If the architecture is the empty list, this corresponds to linear or logistic regression. The optimizer is one of SGD, Momentum, RMS\_Prop.

```

_____learnNN.py — (continued) _____
362 | def create_nn(data, arch, opt, parms):
363 |     """arch is a list of widths of the hidden layers from bottom up.
364 |     opt is an optimizer (one of: SGD, Momentum, RMS_Prop)
365 |     parms is the list of parameters of the optimizer
366 |     returns a neural network with relu activations on hidden layers
367 |     """
368 |     nn = NN(data, optimizer=opt, parms=parms)
369 |     for width in arch:
370 |         nn.add_layer(Linear_complete_layer(nn,width))
371 |         nn.add_layer(ReLU_layer(nn))
372 |     output_size = data.target.frange if data.target.ftype == "categorical"
373 |         else 1
374 |     nn.add_layer(Linear_complete_layer(nn,output_size))
374 |     return nn

```

## 8.6 Plotting Performance

You can plot the performance of various algorithms on the training and validation sets.

Figure 8.1 shows the training and validation performance on the SPECT dataset for the architectures above. The legend give the architecture, the optimizer, the options, and the evaluation dataset. The architecture [] is for logistic regression. Notice how, as the network gets larger the better they fit the training data, but can overfit more as the number of steps increases (probably because the probabilities get more extreme). These figures suggest that early stopping after 200-300 steps might provide best test performance.

```

_____learnNN.py — (continued) _____
376 | from learnLinear import plot_steps
377 | from learnProblem import Evaluate
378 |
379 | # To show plots first choose a criterion to use
380 | crit = Evaluate.log_loss # penalizes overconfident predictions (when wrong)
381 | # crit = Evaluate.accuracy # only considers mode
382 | # crit = Evaluate.squared_loss # penalizes overconfident predictions less

```

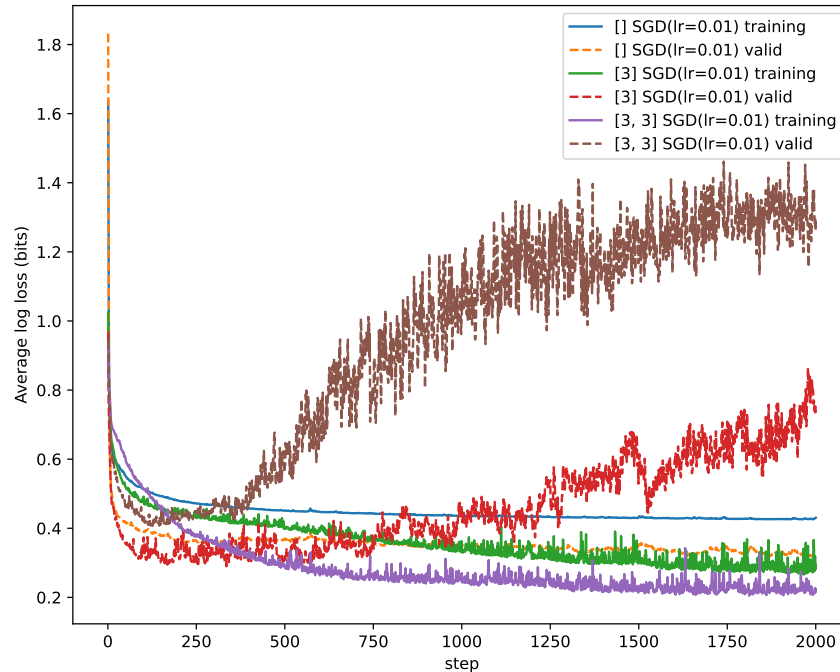


Figure 8.1: Plotting train and validation log loss for various architectures on SPECT dataset. Generated by  
`plot_algs(archs=[[[]],[3],[3,3]], opts=[SGD], lrs=[0.01], num_steps=2000)`

```

383
384 def plot_algs(archs=[[3]], opts=[SGD], lrs=[0.1, 0.01, 0.001, 0.0001],
385               data=data, criterion=crit, num_steps=1000):
386     args = []
387     for arch in archs:
388         for opt in opts:
389             for lr in lrs:
390                 args.append((arch, opt, {'lr': lr}))
391     plot_algs_opts(args, data, criterion, num_steps)
392
393 def plot_algs_opts(args, data=data, criterion=crit, num_steps=1000):
394     """args is a list of (architecture, optimizer, parameters)
395     for each of the corresponding triples it plots the learning rate"""
396     for (arch, opt, parms) in args:
397         nn = create_nn(data, arch, opt, parms)
398         parms_string = ', '.join(f"{p}={v}" for p, v in parms.items())
399         plot_steps(learner = nn, data = data, criterion=crit,
                    num_steps=num_steps,

```

```

400         log_scale=False, legend_label=f"{arch}
         {opt.__name__}({parms_string})")
401
402
403 # first select good learning rates:
404 # plot_algs(archs=[[3]], opts=[SGD], lrs=[0.1, 0.01, 0.001, 0.0001])
405 # plot_algs(archs=[[3]], opts=[Momentum], lrs=[0.1, 0.01, 0.001, 0.0001])
406 # plot_algs(archs=[[3]], opts=[RMS_Prop], lrs=[0.1, 0.01, 0.001, 0.0001])
407 # plot_algs(archs=[[3]], opts=[SGD, Momentum, RMS_Prop], lrs=[0.01])

```

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)
412 # Simplified version: (6000 training instances)
413 # data_mnist = Data_from_file('../MNIST/mnist_train.csv', prob_test=0.9,
         target_index=0, boolean_features=False, target_type="categorical")
414
415 # Full version:
416 # data_mnist = Data_from_files('../MNIST/mnist_train.csv',
         '../MNIST/mnist_test.csv', target_index=0, boolean_features=False,
         target_type="categorical")
417
418 # nn_mnist = NN(data_mnist, learning_rate=0.001)
419 # nn_mnist.add_layer(Linear_complete_layer(nn_mnist, 512));
         nn_mnist.add_layer(ReLU_layer(nn_mnist));
         nn_mnist.add_layer(Linear_complete_layer(nn_mnist, 10))
420 # start_time = time.perf_counter(); nn_mnist.learn(epochs=1,
         batch_size=128); end_time = time.perf_counter(); print("Time:", end_time
         - start_time, "seconds") #1 epoch
421 # determine test error:
422 # data_mnist.evaluate_dataset(data_mnist.test, nn_mnist.predictor,
         Evaluate.accuracy)
423 # Print some random predictions:
424 # 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 8.3** In the definition of `nn3` above, for each of the following, first hypothesize what will happen, then test your hypothesis, then explain whether you testing confirms your hypothesis or not. Test it for more than one data set, and use more than one run for each data set.

- (a) Which fits the data better, having a sigmoid layer or a ReLU layer after the first linear layer?
- (b) Which is faster 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 8.4** For each optimizer, use the validation set to choose the best setting for the hyperparameters, including when to stop, and the parameters of the optimizer (including the learning rate). For the architecture chosen, which optimizer works best? Suggest another architecture which you conjecture would be better on the test set (after hyperparameter optimization). 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 69). 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  $f(V_1 = v_1, V_2 = v_1, \dots, V_k = v_k)$  is stored in  $f[v_1][v_2] \dots [v_k]$ .

If the domain of  $V_i$  is  $[0, \dots, n_i - 1]$  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 conditions; 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.

```

probGraphicalModels.py — (continued)
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

```

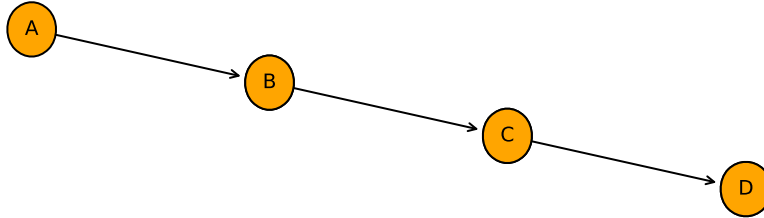
The following creates a topological sort of the nodes, where the parents of a node come before the node in the resulting order. This is based on Kahn's algorithm from 1962.

```

probGraphicalModels.py — (continued)
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.

```

_____probGraphicalModels.py — (continued) _____
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 =
          dict(boxstyle="round4,pad=1.0,rounding_size=0.5",facecolor=facecolor)
72      for var in self.variables: #reversed(self.topological_sort()):
73          for par in self.var2parents[var]:
74              ax.annotate(var.name, par.position, xytext=var.position,
75                          arrowprops={'arrowstyle':'<-'},bbox=bbox,
76                          ha='center', va='center',
77                          fontsize=fontsize)
78      for var in self.variables:
79          x,y = var.position
80          plt.text(x,y,var.name,bbox=bbox,ha='center', va='center',
81                  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.

```

_____probGraphicalModels.py — (continued) _____
81  ##### Simple Example Used for Unit Tests #####

```

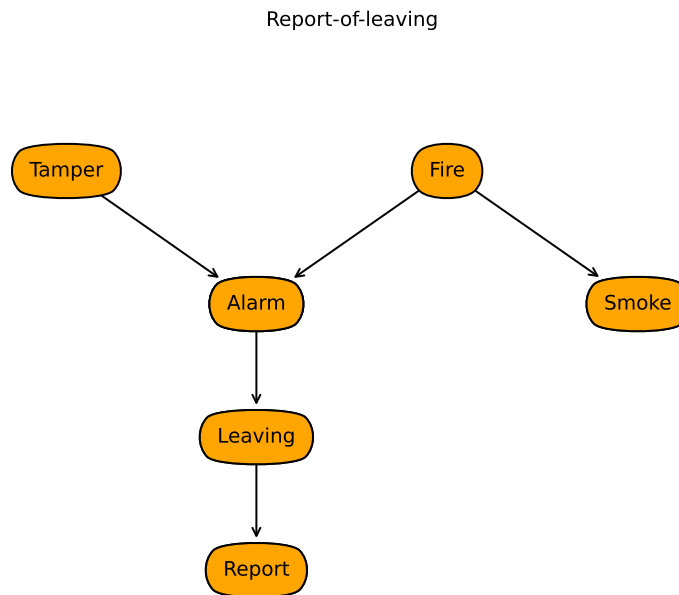


Figure 9.2: The report-of-leaving belief network

```

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 |

```

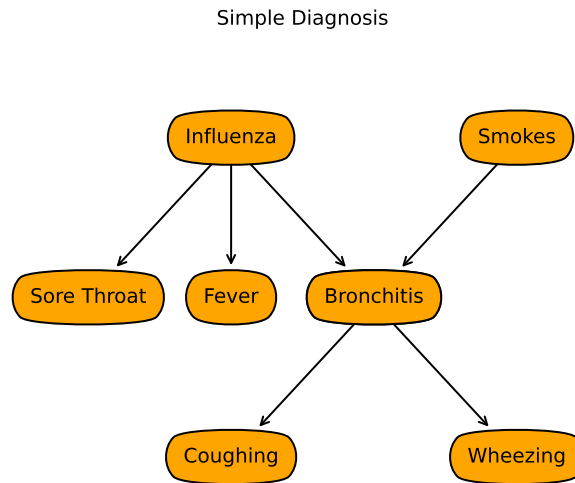


Figure 9.3: Simple diagnosis example; `simple_diagnosis.show()`

```

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]])

```

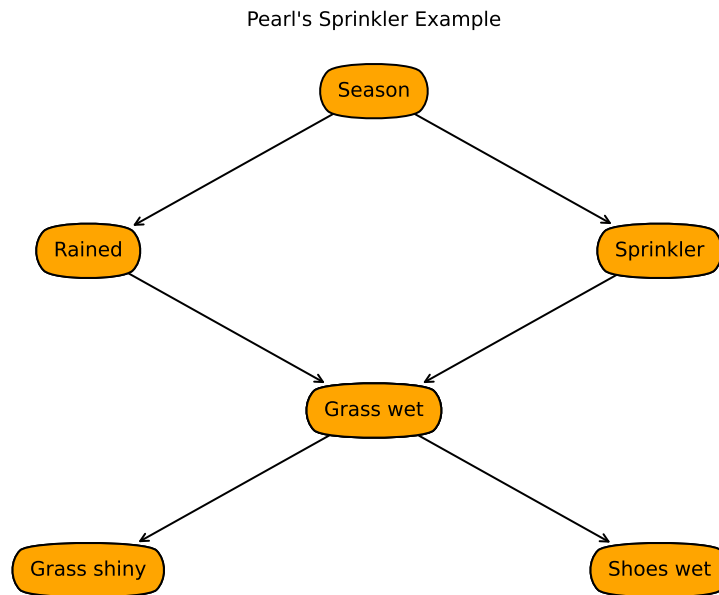


Figure 9.4: The sprinkler belief network

```

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))

```

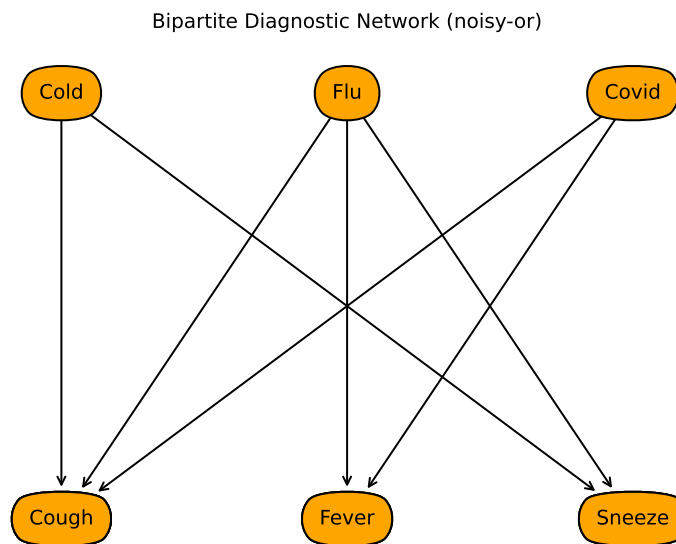


Figure 9.5: A bipartite diagnostic network

```

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()):

```

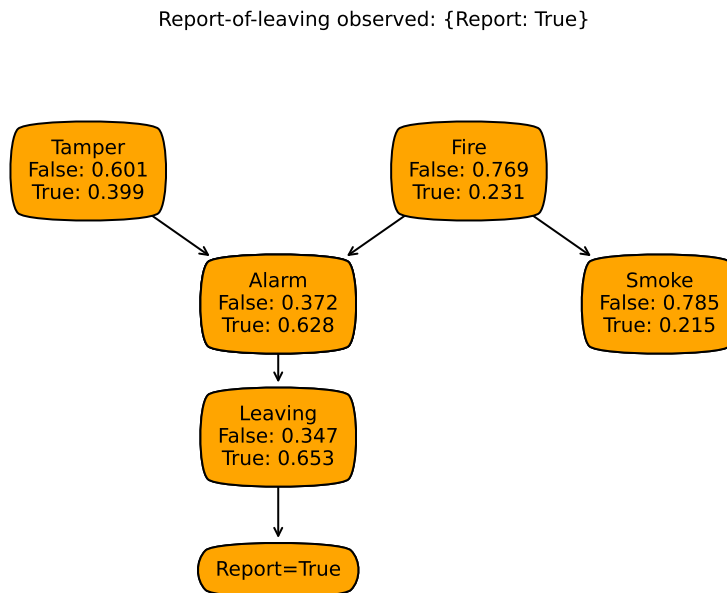


Figure 9.6: The report-of-leaving belief network with posterior distributions

```

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 208).

```

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.

**Exercise 9.2** This code adds to the cache only after splitting. Implement a variant that caches after forgetting. (What can the cache start with?) Which version works better? Compare some measure of the search tree and the space used. Try other alternatives of what to cache; which method works best?

## 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.

```

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 is assigning a value to a single variable.

---

```

probVE.py — (continued)
43 def project_observations(self,factor,obs):
44     """Returns the resulting factor after observing obs
45
46     obs is a dictionary of {variable:value} pairs.
47     """
48     if any((var in obs) for var in factor.variables):
49         # a variable in factor is observed
50         return FactorObserved(factor,obs)
51     else:
52         return factor

```

```

53
54 def eliminate_var(self,factors,var):
55     """Eliminate a variable var from a list of factors.
56     Returns a new set of factors that has var summed out.
57     """
58     self.display(2,"eliminating ",str(var))
59     contains_var = []
60     not_contains_var = []
61     for fac in factors:
62         if var in fac.variables:
63             contains_var.append(fac)
64         else:
65             not_contains_var.append(fac)
66     if contains_var == []:
67         return factors
68     else:
69         newFactor = FactorSum(var,contains_var)
70         self.display(2,"Multiplying:",[str(f) for f in contains_var])
71         self.display(2,"Creating factor:", newFactor)
72         self.display(3, newFactor.to_table()) # factor in detail
73         not_contains_var.append(newFactor)
74     return not_contains_var
75
76 from probGraphicalModels import bn_4ch, A,B,C,D
77 bn_4chv = VE(bn_4ch)
78 ## bn_4chv.query(A,{})
79 ## bn_4chv.query(D,{})
80 ## InferenceMethod.max_display_level = 3 # show more detail in displaying
81 ## InferenceMethod.max_display_level = 1 # show less detail in displaying
82 ## bn_4chv.query(A,{D:True})
83 ## bn_4chv.query(B,{A:True,D:False})
84
85 from 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

```

```

102 | vediag = VE(bn_lr1)
103 | ## vediag.query(Cough,{})
104 | ## vediag.query(Cold,{Cough:1,Sneeze:0,Fever:1})
105 | ## vediag.query(Flu,{Cough:0,Sneeze:1,Fever:1})
106 | ## 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.

```

```

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

```

### Exercise 9.3

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
89             in nvar.domain})
90         self.display(2,val,end="\t")
91         if nvar in obs and obs[nvar] != val:
92             rejected = True
93             self.display(2,"Rejected")
94             break
95         sample[nvar] = val
96         if not rejected:
97             counts[sample[qvar]] += 1
98             self.display(2,"Accepted")
99         tot = sum(counts.values())
100         # As well as the distribution we also include raw counts
101         dist = {c:v/tot if tot>0 else 1/len(qvar.domain) for (c,v) in
102             counts.items()}
103         dist["raw_counts"] = counts
104         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                                 v in nvar.domain})
137                 self.display(2,val,end="\t")
138                 sample[nvar] = val
139             counts[sample[qvar]] += weight
140             self.display(2,weight)
141         tot = sum(counts.values())
142         # as well as the distribution we also include the raw counts
143         dist = {c:v/tot for (c,v) in counts.items()}
144         dist["raw_counts"] = counts
145         return dist

```

**Exercise 9.4** 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 adjust 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 proposal 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 = [{i} 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             else:
177                 for part in particles:
178                     part[nvar] = sample_one({v:fac.get_value(**part,
179                                                             nvar:v)}
180                                             for v in nvar.domain))
181                 self.display(2,part[nvar],end="\t")
182             counts = {val:0 for val in qvar.domain}
183             for part in particles:
184                 counts[part[qvar]] += 1
185             tot = sum(counts.values())
186             # as well as the distribution we also include the raw counts
187             dist = {c:v/tot for (c,v) in counts.items()}
188             dist["raw_counts"] = counts
189             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]

```

```

204         result.append(particles[index])
205     return result

```

### 9.9.6 Examples

```

_____probStochSim.py — (continued)_____
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}

```

```

285         for val in var.domain:
286             sample[var] = val
287             for fac in var_to_factors[var]: # Markov blanket
288                 vardist[val] *= fac.get_value(sample)
289             sample[var] = sample_one(vardist)
290         if i >= burn_in:
291             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 9.5** 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 9.6** 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

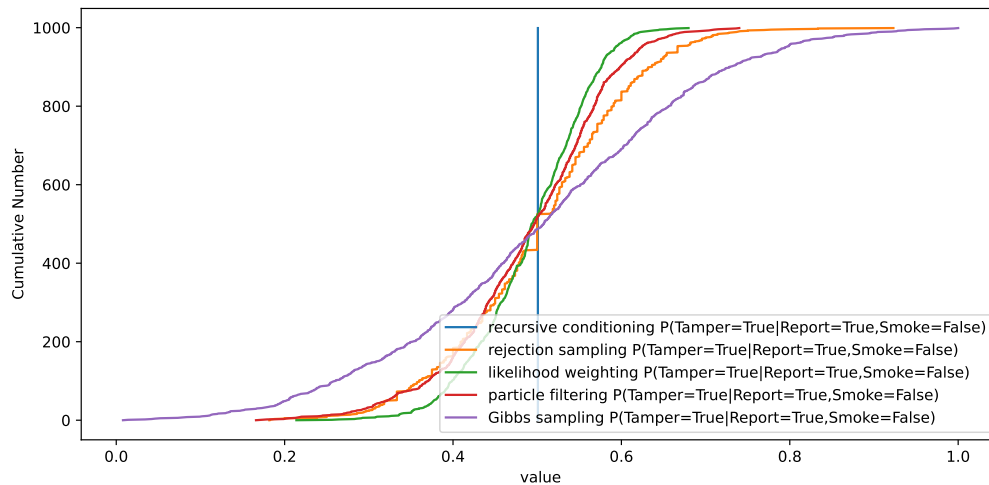


Figure 9.7: Cumulative distribution of the prediction of various models for  $P(\text{Tamper}=\text{True} \mid \text{report} \wedge \neg \text{smoke})$

Gaussian) may not hold for cases where the predictions are bounded and often skewed.

It is more appropriate to plot the distribution of predictions over multiple runs. The `plot_stats` method plots the prediction of a particular variable (or for the partition function) for a number of runs of the same algorithm. On the  $x$ -axis, is the prediction of the algorithm. On the  $y$ -axis is the number of runs with prediction less than or equal to the  $x$  value. Thus this is like a cumulative distribution over the predictions, but with counts on the  $y$ -axis.

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

That variable *what* contains the query variable, or 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.

probStochSim.py — (continued)

```

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_iterations is the number of runs that are plotted
324     **queryargs is the arguments to query (often number_samples for
        sampling methods)
325     """
326     plt.ion()
327     plt.xlabel("value")
328     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},
      number_samples=1000)
365 # 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.

```

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.

```

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},
38         # mic 1
39         'm2':{'middle':midMic, 'c1':farMic, 'c2':closeMic, 'c3':farMic}, #
40         # mic 2
41         'm3':{'middle':midMic, 'c1':farMic, 'c2':farMic, 'c3':closeMic}} #
42         # 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.

```

43 # trans specifies the dynamics
44 # trans[i] is the distribution over states resulting from state i
45 # trans[i][j] gives P(S=j | S=i)
46 sm=0.7; mmc=0.1 # transition probabilities when in middle
47 sc=0.8; mcm=0.1; mcc=0.05 # transition probabilities when in a corner
48 trans1 = {'middle':{'middle':sm, 'c1':mmc, 'c2':mmc, 'c3':mmc}, # was in
49         # middle
50         'c1':{'middle':mcm, 'c1':sc, 'c2':mcc, 'c3':mcc}, # was in corner
51         # 1
52         'c2':{'middle':mcm, 'c1':mcc, 'c2':sc, 'c3':mcc}, # was in corner
53         # 2
54         'c3':{'middle':mcm, 'c1':mcc, 'c2':mcc, 'c3':sc}} # was in corner
55         # 3

```

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

```

56 # initially we have a uniform distribution over the animal's state
57 indist1 = {st:1.0/len(states1) for st in states1}
58
59 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.

```

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

```

**Exercise 9.7** The representation assumes that there are a list of Boolean observations. Extend the representation so that the each observation variable can have multiple discrete values. You need to choose a representation for the model, and change the algorithm.

### 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
18 |       action.
19 |       Instead of the transition probability, it has a function act2trans
20 |       from action to transition probability.
21 |       Any algorithms need to select the transition probability according
22 |       to the action.
23 |    """

```

```

22     def __init__(self, states, obsvars, pobs, act2trans, indist):
23         self.act2trans = act2trans
24         HMM.__init__(self, states, obsvars, pobs, None, indist)
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

```

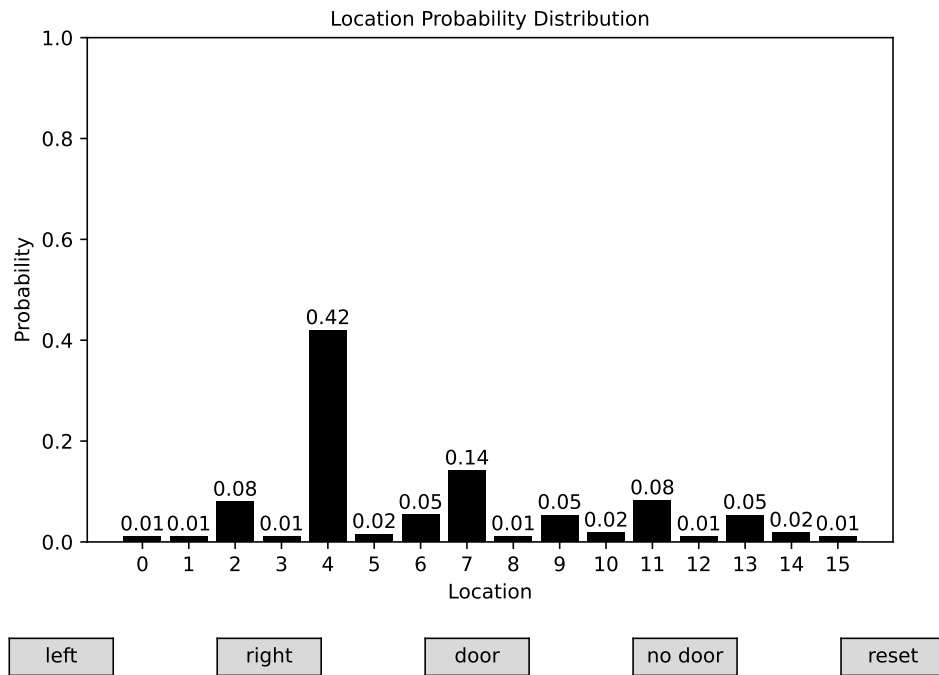


Figure 9.8: Localization GUI after observing a door, moving right, observing no door, moving right, and observing a door.

```

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")
66     left_but.label.set_fontsize(self.fontsize)
67     left_but.on_clicked(self.left)
68     right_but = Button(plt.axes([0.25,0.02,0.1,0.05]), "right")
69     right_but.label.set_fontsize(self.fontsize)
70     right_but.on_clicked(self.right)
71     door_but = Button(plt.axes([0.45,0.02,0.1,0.05]), "door")
72     door_but.label.set_fontsize(self.fontsize)
73     door_but.on_clicked(self.door)
74     nodoor_but = Button(plt.axes([0.65,0.02,0.1,0.05]), "no door")
75     nodoor_but.label.set_fontsize(self.fontsize)
76     nodoor_but.on_clicked(self.nodoor)
77     reset_but = Button(plt.axes([0.85,0.02,0.1,0.05]), "reset")
78     reset_but.label.set_fontsize(self.fontsize)
79     reset_but.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
104    def right(self, event):
105        self.loc_filt.go("right")
106        self.draw_dist()
107
108    def door(self, event):
109        self.loc_filt.observe({'door': True})
110        self.draw_dist()
111
112    def nodoor(self, event):
113        self.loc_filt.observe({'door': False})
114        self.draw_dist()
115
116    def reset(self, event):
117        self.loc_filt.state_dist = {i:1/16 for i in range(16)}
118        self.draw_dist()
119
120    # Show_Localization(hmm_16pos)
121    # Show_Localization(hmm_16pos, fontsize=15) # for demos - enlarge window
122
123    if __name__ == "__main__":
124        print("Try: Show_Localization(hmm_16pos)")

```

### 9.10.3 Particle Filtering for HMMs

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

probHMM.py — (continued)

```

114 from display import Displayable
115 from probStochSim import resample

```

```

116
117 class HMMparticleFilter(Displayable):
118     def __init__(self,hmm,number_particles=1000):
119         self.hmm = hmm
120         self.particles = [sample_one(hmm.indist)
121                             for i in range(number_particles)]
122         self.weights = [1 for i in range(number_particles)]
123
124     def filter(self, obsseq):
125         """returns the state distribution following the sequence of
126         observations in obsseq using particle filtering.
127
128         Note that it first advances time.
129         This is what is required if it is called after previous filtering.
130         If that is not what is wanted initially, do an observe first.
131         """
132         for obs in obsseq:
133             self.advance() # advance time
134             self.observe(obs) # observe
135             self.resample_particles()
136             self.display(2,"After observing", str(obs),
137                         "state distribution:",
138                         self.histogram(self.particles))
139             self.display(1,"Final state distribution:",
140                         self.histogram(self.particles))
141         return self.histogram(self.particles)
142
143     def advance(self):
144         """advance to the next time.
145         This assumes that all of the weights are 1."""
146         self.particles = [sample_one(self.hmm.trans[st])
147                             for st in self.particles]
148
149     def observe(self, obs):
150         """reweighs the particles to incorporate observations obs"""
151         for i in range(len(self.particles)):
152             for obv in obs:
153                 if obs[obv]:
154                     self.weights[i] *= self.hmm.pobs[obv][self.particles[i]]
155                 else:
156                     self.weights[i] *=
157                         1-self.hmm.pobs[obv][self.particles[i]]
158
159     def histogram(self, particles):
160         """returns list of the probability of each state as represented by
161         the particles"""
162         tot=0
163         hist = {st: 0.0 for st in self.hmm.states}
164         for (st,wt) in zip(self.particles,self.weights):
165             hist[st]+=wt

```

```

163         tot += wt
164         return {st:hist[st]/tot for st in hist}
165
166     def resample_particles(self):
167         """resamples to give a new set of particles."""
168         self.particles = resample(self.particles, self.weights,
169                                   len(self.particles))
169         self.weights = [1] * len(self.particles)

```

The following are some queries for *hmm1*.

```

_____probHMM.py — (continued)_____
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':0}, {'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 9.8** A form of importance sampling can be obtained by not resampling. Is it better or worse than particle filtering? Hint: you need to think about how they can be compared. Is the comparison different if there are more states than particles?

**Exercise 9.9** Extend the particle filtering code to continuous variables and observations. In particular, suppose the state transition is a linear function with Gaussian noise of the previous state, and the observations are linear functions with Gaussian noise of the state. You may need to research how to sample from a Gaussian distribution (or use Python's random library) .

### 9.10.4 Generating Examples

The following code is useful for generating examples.

```

_____probHMM.py — (continued)_____
182 def simulate(hmm,horizon):
183     """returns a pair of (state sequence, observation sequence) of length
184         horizon.
185     for each time t, the agent is in state_sequence[t] and
186     observes observation_sequence[t]
187     """
188     state = sample_one(hmm.indist)
189     obsseq=[]

```

```

189     stateseq=[]
190     for time in range(horizon):
191         stateseq.append(state)
192         newobs =
193             {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
194              for obs in hmm.obsvars}
195         obsseq.append(newobs)
196         state = sample_one(hmm.trans[state])
197     return stateseq,obsseq
198
199 def simobs(hmm,stateseq):
200     """returns observation sequence for the state sequence"""
201     obsseq=[]
202     for state in stateseq:
203         newobs =
204             {obs:sample_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})
205              for obs in hmm.obsvars}
206         obsseq.append(newobs)
207     return obsseq
208
209 def create_eg(hmm,n):
210     """Create an annotated example for horizon n"""
211     seq,obs = simulate(hmm,n)
212     print("True state sequence:",seq)
213     print("Sequence of observations:\n",obs)
214     hmmfilter = HMMVEfilter(hmm)
215     dist = hmmfilter.filter(obs)
216     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.

```

_____probDBN.py — Dynamic belief networks_____
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:

```

```

42     (x,y) = position
43     position = (x-0.3, y)
44     var_prev = DBNvariable(name, domain, index='prev', position=position)
45     var_now.previous = var_prev
46     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

```

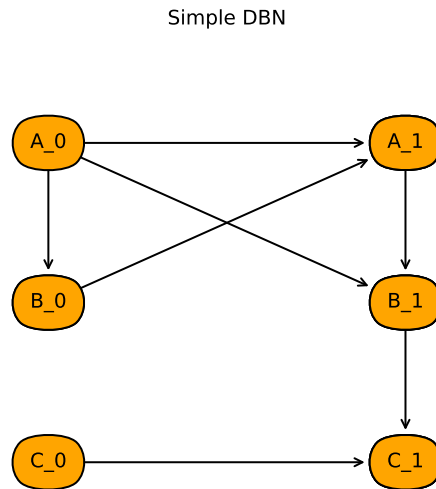


Figure 9.9: Simple dynamic belief network (dbn1.show())

```

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):
85         self.title = title
86         self.vars_now = vars_now
87         self.vars_prev = [v.previous for v in vars_now]
88         self.transition_factors = transition_factors
89         self.init_factors = init_factors
90         self.var_index = {} # var_index[v] is the index of variable v
91         for i,v in enumerate(vars_now):
92             self.var_index[v]=i
93
94     def show(self):
95         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]])

```

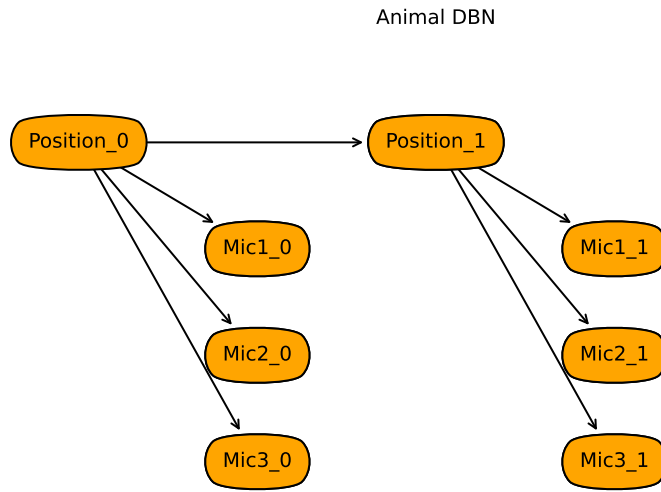


Figure 9.10: Animal dynamic belief network (dbn\_an.show())

```

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

```

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))
116 | Mic1_0,Mic1_1 = variable_pair("Mic1", position=(0.6,0.6))
117 | Mic2_0,Mic2_1 = variable_pair("Mic2", position=(0.6,0.4))
118 | Mic3_0,Mic3_1 = variable_pair("Mic3", position=(0.6,0.2))
119 |
120 | # conditional probabilities - see hmm for the values of sm,mmc, etc
121 | ppos = Prob(Pos_1, [Pos_0],
122 |             [[sm, mmc, mmc, mmc], #was in middle
123 |              [mcm, sc, mcc, mcc], #was in corner 1
124 |              [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

```

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

```

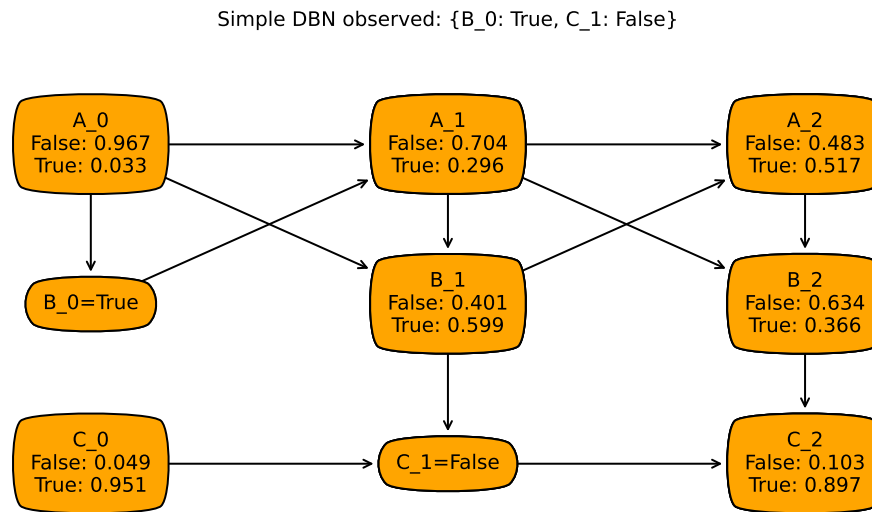


Figure 9.11: Simple dynamic belief network (dbn1) horizon 2

```

166 def pos(self, var, index):
167     minx = min(x for (x,y) in (var.position for var in
168         self.dbn.vars_now))-1e-6
169     maxx = max(x for (x,y) in (var.position for var in
170         self.dbn.vars_now))
171     width = maxx-minx
172     xo,yo = var.position
173     xi = index/(self.horizon+1)+(xo-minx)/width/(self.horizon+1)/2
174     return (xi, yo)

```

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})

```

```

183 # Plot Distributions:
184 # bna = BNfromDBN(dbn_an,5) # animal belief network with horizon 5
185 # dra = ProbRC(bna)
186 # dra.show_post(obs =
    {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
                   observations"
201         self.current_obs.update(obs) # note 'update' is a dict method
202
203     def query(self,var):
204         """returns the posterior probability of current variable var"""
205         return
206             VE(GraphicalModel(self.dbn.title,self.dbn.vars_now,self.current_factors)
207               ).query(var,self.current_obs)
208
209     def advance(self):
210         """advance to the next time"""
211         prev_factors = [self.make_previous(fac) for fac in
212                         self.current_factors]
213         prev_obs = {var.previous:val for var,val in
214                     self.current_obs.items()}
215         two_stage_factors = prev_factors + self.dbn.transition_factors
216         self.current_factors =
217             self.elim_vars(two_stage_factors,self.dbn.vars_prev,prev_obs)
218         self.current_obs = {}
219
220     def make_previous(self,fac):
221         """Creates new factor from fac where the current variables in fac
222         are renamed to previous variables.
223         """
224         return FactorRename(fac, {var.previous:var for var in
225                                fac.variables})

```

```

221
222     def elim_vars(self,factors, vars, obs):
223         for var in vars:
224             if var in obs:
225                 factors = [self.project_observations(fac,obs) for fac in
                             factors]
226             else:
227                 factors = self.eliminate_var(factors, var)
228         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)]
```

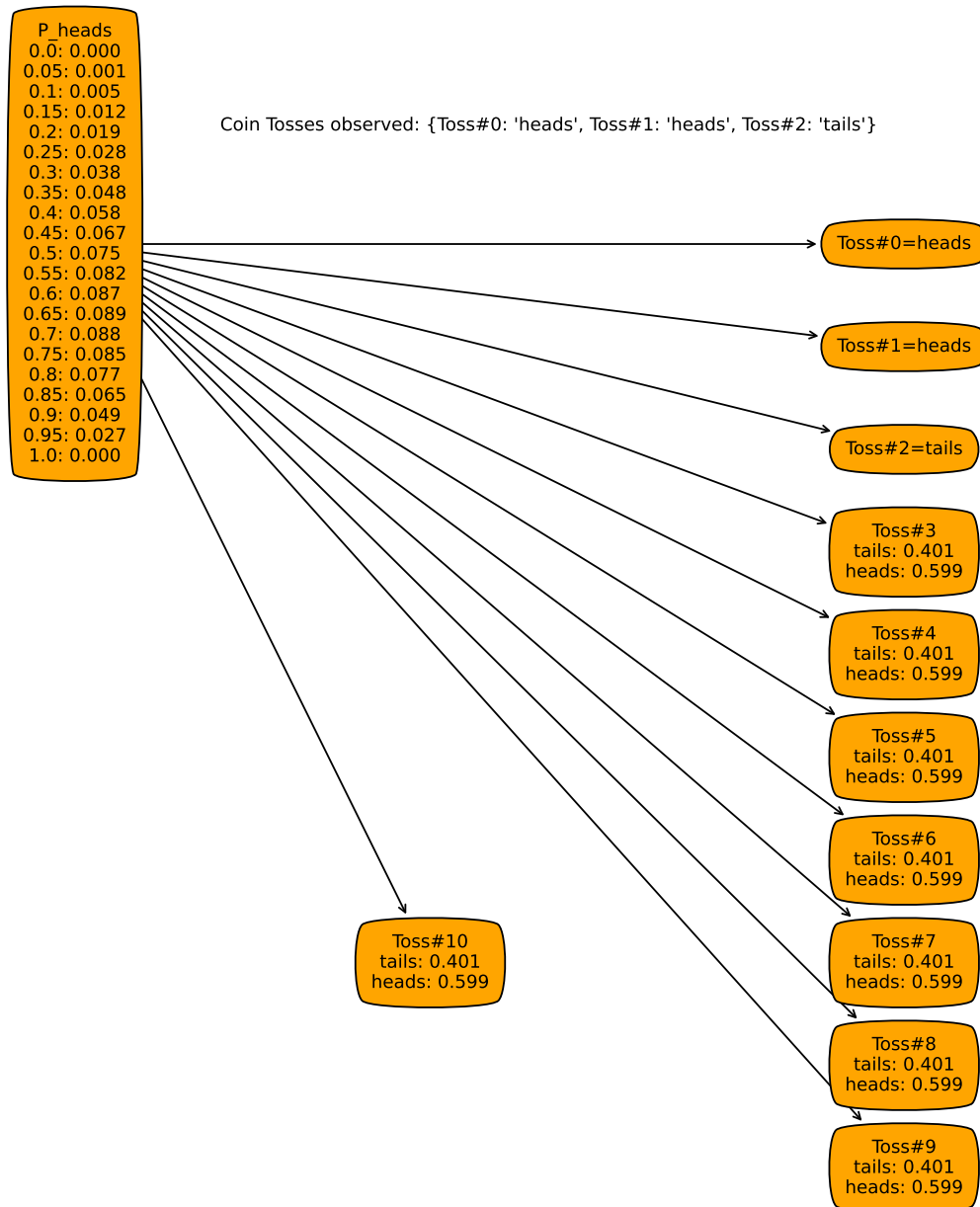


Figure 10.1: coinTossBN after observing heads, heads, tails

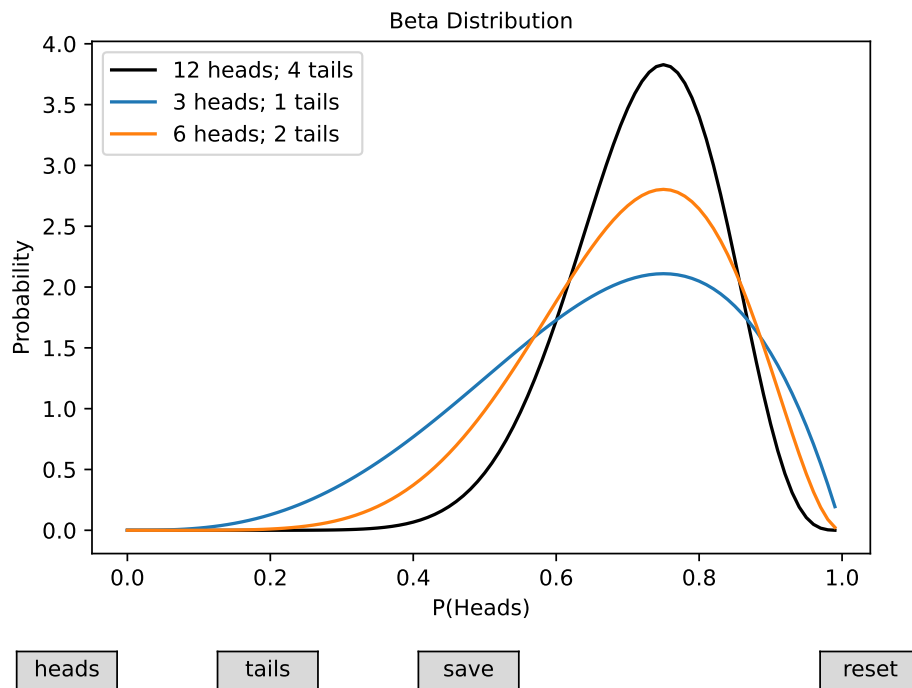


Figure 10.2: Beta distribution after some observations

```

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_butt = Button(plt.axes([0.05,0.02,0.1,0.05]), "heads")
60     heads_butt.label.set_fontsize(self.fontsize)
61     heads_butt.on_clicked(self.heads)
62     tails_butt = Button(plt.axes([0.25,0.02,0.1,0.05]), "tails")
63     tails_butt.label.set_fontsize(self.fontsize)
64     tails_butt.on_clicked(self.tails)
65     save_butt = Button(plt.axes([0.45,0.02,0.1,0.05]), "save")
66     save_butt.label.set_fontsize(self.fontsize)
67     save_butt.on_clicked(self.save)
68     reset_butt = Button(plt.axes([0.85,0.02,0.1,0.05]), "reset")
69     reset_butt.label.set_fontsize(self.fontsize)
70     reset_butt.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
111
112    if __name__ == "__main__":
113        print("Try: Show_Beta(100)")

```

## 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,"
                        Stable=",stable)
84     return stable
85
86 def show_classes(self):
87     """sorts the data by the class and prints in order.
88     For visualizing small data sets
89     """
90     class_examples = [[] for i in range(self.num_classes)]
91     for eg in self.dataset.train:
92         class_examples[self.class_of_eg(eg)].append(eg)
93     print("Class", "Example", sep='\t')
94     for cl in range(self.num_classes):
95         for eg in class_examples[cl]:
96             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 dif-

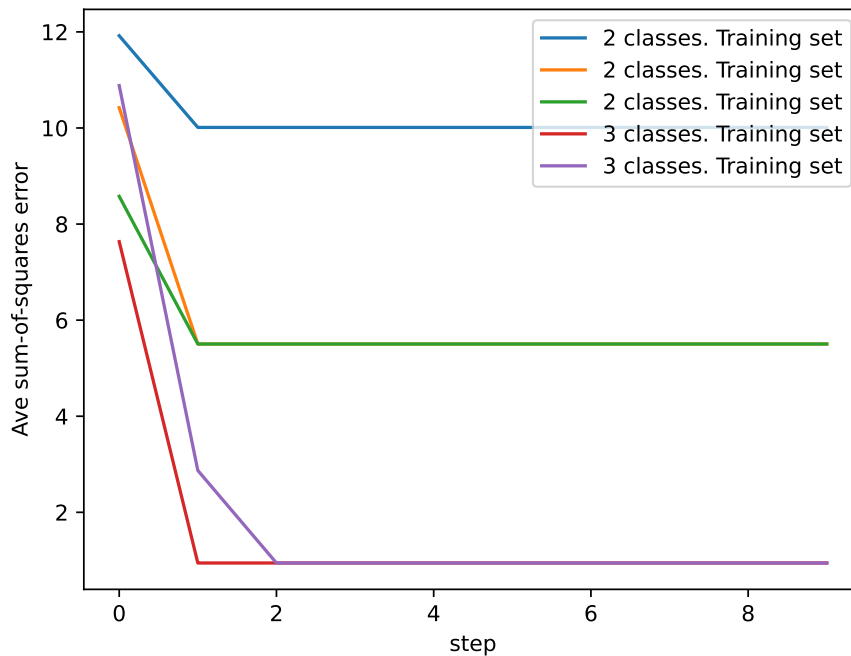


Figure 10.3: k-means plotting error.

ferent 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 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))

```



```

109         if self.dataset.test:
110             test_errors.append(
111                 sum(self.distance(self.class_of_eg(eg), eg)
112                     for eg in self.dataset.test)
113                 / len(self.dataset.test))
114             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:

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

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

## 10.3 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.  $\text{val}$  is a value of a feature.

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

- $\text{class\_counts}$  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.,

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

- $\text{feature\_counts}$  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
                                distribution
33         tpl_class_dist = random_dist(self.num_classes)
34         for cl in range(self.num_classes):
35             class_counts[cl] += tpl_class_dist[cl]
36             for (ind, feat) in enumerate(self.dataset.input_features):
37                 feature_counts[ind][feat(tple)][cl] += tpl_class_dist[cl]
38         return class_counts, feature_counts

```

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

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

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

```

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

```

*learn* does  $n$  steps of EM:

```

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

```

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

```

learnEM.py — (continued)
57 def show_class(self, c):

```

```

58     """sorts the data by the class and prints in order.
59     For visualizing small data sets
60     """
61     sorted_data =
62         sorted((self.prob(tpl,self.class_counts,self.feature_counts)[c],
63                ind, # preserve ordering for equal
64                  probabilities
65                  tpl)
66               for (ind,tpl) in enumerate(self.dataset.train))
67     for cc,r,tpl in sorted_data:
68         print(cc,*tpl,sep='\t')
```

The following are for evaluating the classes.

The probability of a tuple can be evaluated by marginalizing over the classes:

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

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).

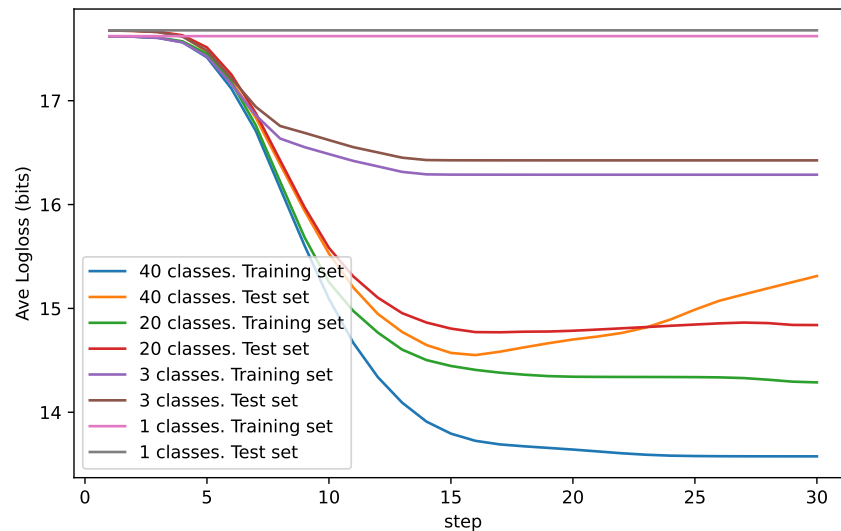


Figure 10.4: EM plotting error.

```

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={}):
```

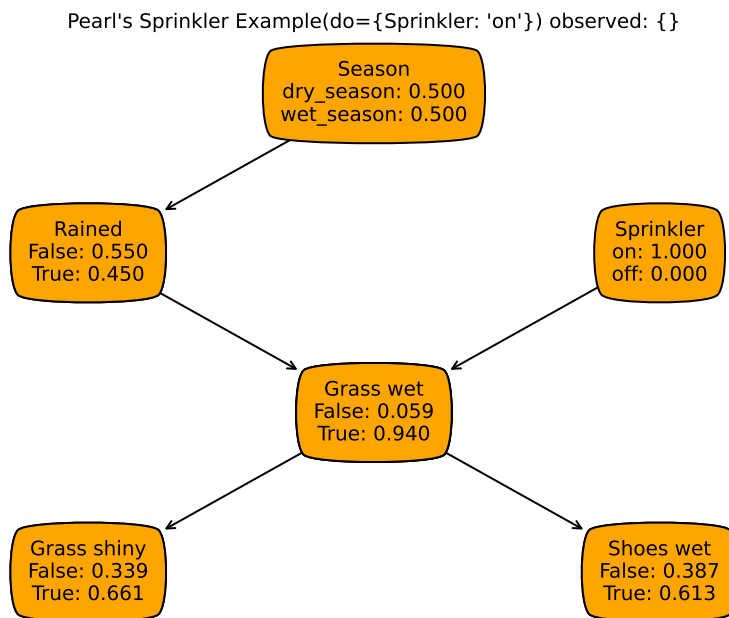


Figure 11.1: The sprinkler belief network with  $\text{do}=\{\text{Sprinkler: "on"}\}$ .

```

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.

```

_____probDo.py — (continued) _____
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: {}

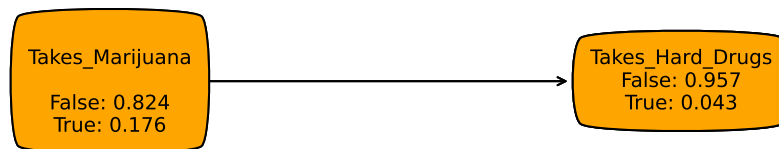


Figure 11.2: Does taking marijuana lead to hard drugs: observable variables

```

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

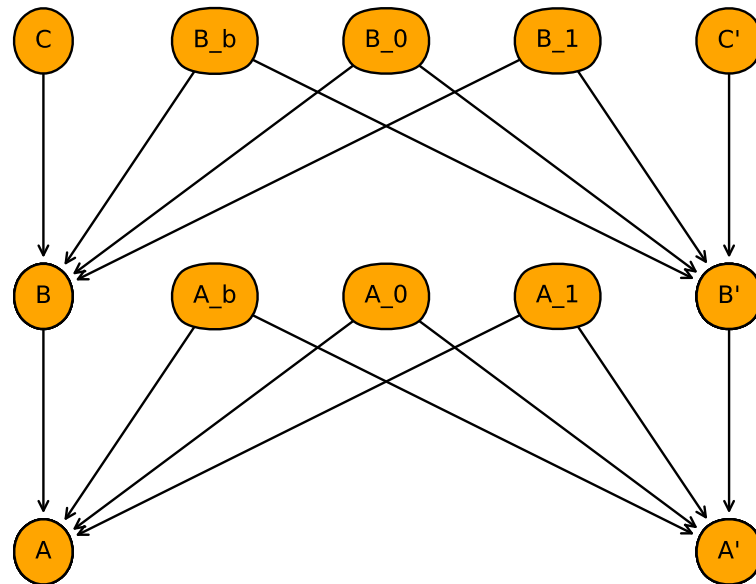


Figure 11.3:  $C \rightarrow B \rightarrow A$  belief network for “what if  $C$ ”. Figure generated by `cbaCounter.show()`

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,  $C_{\text{prime}}$  should be conditioned on. Conditioning on  $C_{\text{prime}}$  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(Bprime, [], [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:

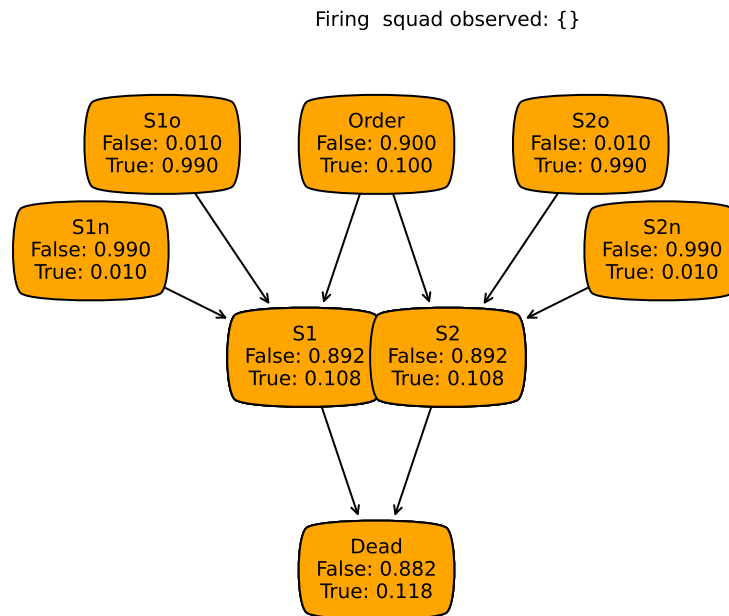


Figure 11.4: Firing squad belief network (figure obtained from `fsq.show_post({})`)

- 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.

**Exercise 11.3** Choose two assignments to values to each of  $a_b$ ,  $a_0$  and  $a_1$  using  $a \equiv a_b \vee (\neg b \wedge a_0) \vee (b \wedge a_1)$ , and a counterfactual query such that (a) the two assignments cannot be distinguished by observations or by interventions, and (b) the predictions for the query differ by an arbitrarily large amount (differ by  $1 - \epsilon$  for a small value of  $\epsilon$ , such as  $\epsilon = 0.1$ ).

### 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,
102                              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 11.4** 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 11.5** Create the network for “what if the order was or was not given”. Give the probabilities of the following counterfactuals:

- (a) The prisoner is dead; what is the probability that the prisoner would be dead if the order was not given?
- (b) 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 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],

```

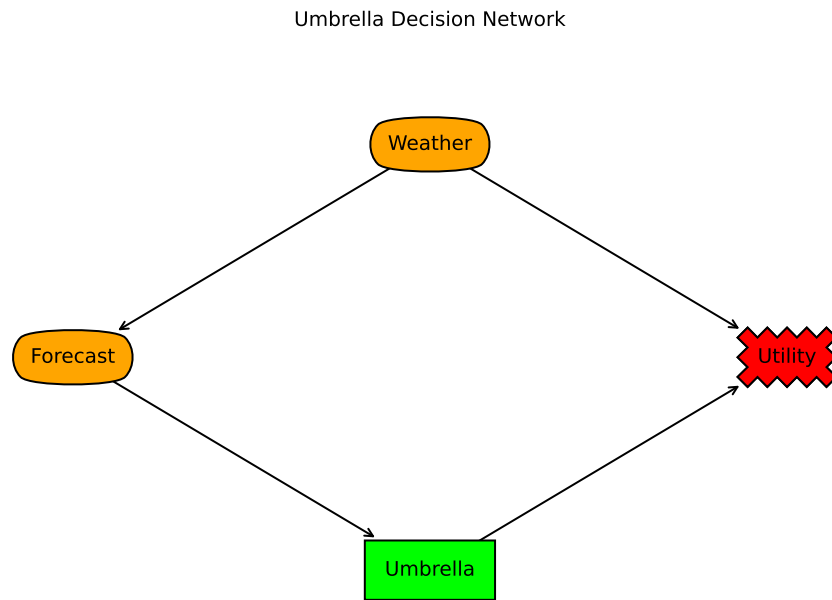


Figure 12.1: The umbrella decision network. Figure generated by `umbrella_dn.show()`

```

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)

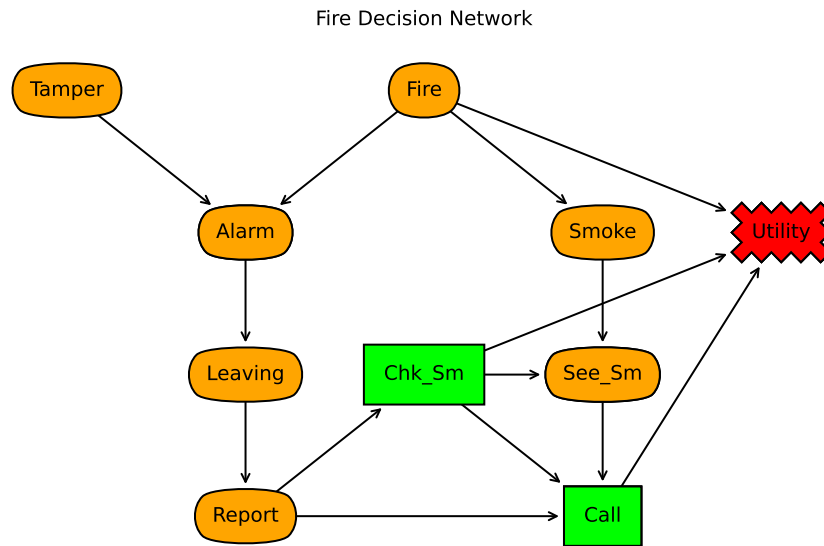
```

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)

```

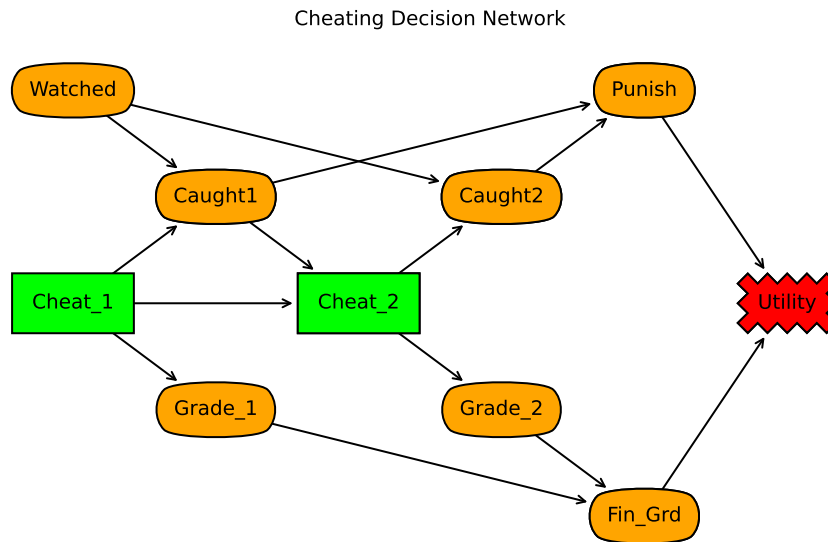
### Cheating Decision Network

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, 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})

```



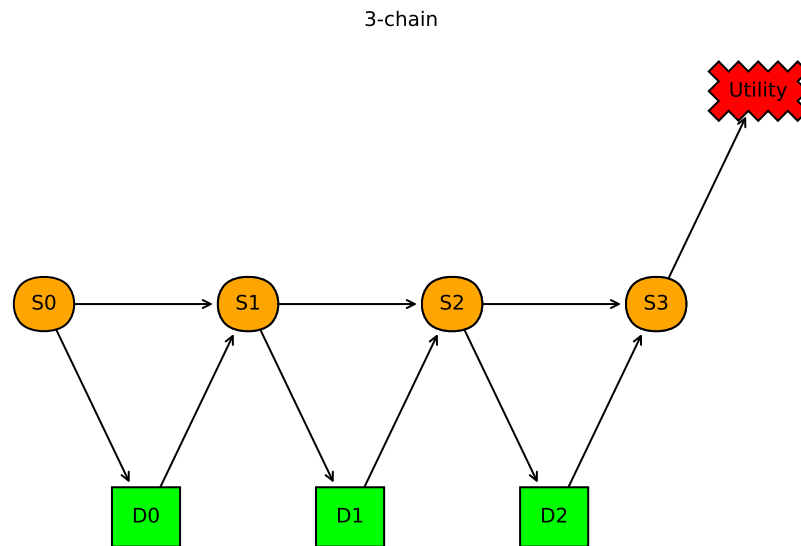


Figure 12.4: A decision network that is a chain of 3 decisions (`ch3.show()`)

```

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 220).

```

340 | def rc(self, context, factors, split_order):
341 |     """ returns the number sum_{split_order} prod_{factors} given
342 |         assignments in context
343 |     context is a variable:value dictionary
344 |     factors is a set of factors
345 |     split_order is a list of variables in factors that are not in
346 |         context
347 |     """
348 |     self.display(3,"calling rc,", (context,factors))
349 |     ce = (frozenset(context.items()), frozenset(factors)) # key for the
350 |         cache entry
351 |     if ce in self.cache:
352 |         self.display(2,"rc cache lookup", (context,factors))
353 |         return self.cache[ce]
354 |     # if not factors: # no factors; needed if you don't have forgetting
355 |     and caching
356 |     # return 1
357 |     elif vars_not_in_factors := {var for var in context
358 |         if not any(var in fac.variables for
359 |             fac in factors)}:
360 |         # forget variables not in any factor
361 |         self.display(3,"rc forgetting variables", vars_not_in_factors)
362 |         return self.rc({key:val for (key,val) in context.items()
363 |             if key not in vars_not_in_factors},
364 |             factors, split_order)
365 |     elif to_eval := {fac for fac in factors if
366 |         fac.can_evaluate(context)}:
367 |         # evaluate factors when all variables are assigned
368 |         self.display(3,"rc evaluating factors",to_eval)
369 |         val = math.prod(fac.get_value(context) for fac in to_eval)
370 |         if val == 0:
371 |             return 0
372 |         else:
373 |             return val * self.rc(context, {fac for fac in factors if fac
374 |                 not in to_eval}, split_order)
375 |     elif len(comp := connected_components(context, factors,
376 |         split_order)) > 1:
377 |         # there are disconnected components
378 |         self.display(2,"splitting into connected components",comp)
379 |         return(math.prod(self.rc(context,f,eo) for (f,eo) in comp))
380 |     else:
381 |         assert split_order, f"split_order empty rc({context},{factors})"
382 |         var = split_order[0]
383 |         self.display(3, "rc branching on", var)

```

```

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.

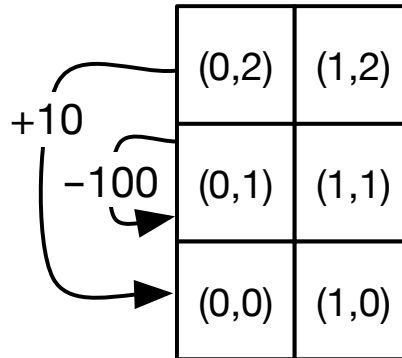


Figure 12.5: Tiny game

(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

```