#### 1

Python code for Artificial Intelligence

# Contents

[Contents](#_bookmark0) 3

1. [Python for Artificial Intelligence](#_bookmark1) 7

[1.1 Why Python?](#_bookmark3) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

[1.2 Getting Python](#_bookmark4) . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

[1.3 Running Python](#_bookmark6) . . . . . . . . . . . . . . . . . . . . . . . . . . 8

[1.4 Pitfalls](#_bookmark7) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

[1.5 Features of Python](#_bookmark8) . . . . . . . . . . . . . . . . . . . . . . . . . 9

* 1. [Useful Libraries](#_bookmark14) 13
  2. [Utilities](#_bookmark15) 14
  3. [Testing Code](#_bookmark20) 17

1. [Agents and Control](#_bookmark21) 19
   1. [Representing Agents and Environments](#_bookmark23) 19
   2. [Paper buying agent and environment](#_bookmark25) 20
   3. [Hierarchical Controller](#_bookmark28) 23
2. [Searching for Solutions](#_bookmark33) 31
   1. [Representing Search Problems](#_bookmark35) 31
   2. [Generic Searcher and Variants](#_bookmark42) 38
   3. [Branch-and-bound Search](#_bookmark48) 44
3. [Reasoning with Constraints](#_bookmark50) 49
   1. [Constraint Satisfaction Problems](#_bookmark52) 49
   2. [Solving a CSP using Search](#_bookmark58) 56
   3. [Consistency Algorithms](#_bookmark59) 58

[3](#_TOC_250000)

1. Contents
   1. [Solving CSPs using Stochastic Local Search](#_bookmark65) 63
2. [Propositions and Inference](#_bookmark74) 73
   1. [Representing Knowledge Bases](#_bookmark76) 73
   2. [Bottom-up Proofs](#_bookmark79) 75
   3. [Top-down Proofs](#_bookmark81) 77
   4. [Assumables](#_bookmark82) 78
3. [Planning with Certainty](#_bookmark85) 81
   1. [Representing Actions and Planning Problems](#_bookmark87) 81
   2. [Forward Planning](#_bookmark92) 85
   3. [Regression Planning](#_bookmark98) 89
   4. [Planning as a CSP](#_bookmark101) 92
   5. [Partial-Order Planning](#_bookmark103) 95
4. [Supervised Machine Learning](#_bookmark107) 103
   1. [Representations of Data and Predictions](#_bookmark109) 103
   2. [Learning With No Input Features](#_bookmark115) 113
   3. [Decision Tree Learning](#_bookmark119) 116
   4. [Cross Validation and Parameter Tuning](#_bookmark121) 120
   5. [Linear Regression and Classification](#_bookmark123) 122
   6. [Deep Neural Network Learning](#_bookmark127) 128
   7. [Boosting](#_bookmark131) 133
5. [Reasoning Under Uncertainty](#_bookmark133) 137
   1. [Representing Probabilistic Models](#_bookmark135) 137
   2. [Factors](#_bookmark138) 138
   3. [Graphical Models](#_bookmark144) 143
   4. [Variable Elimination](#_bookmark147) 145
   5. [Stochastic Simulation](#_bookmark149) 147
   6. [Markov Chain Monte Carlo](#_bookmark157) 155
   7. [Hidden Markov Models](#_bookmark159) 157
   8. [Dynamic Belief Networks](#_bookmark162) 163
6. [Planning with Uncertainty](#_bookmark166) 167
   1. [Decision Networks](#_bookmark168) 167
   2. [Markov Decision Processes](#_bookmark174) 172
   3. [Value Iteration](#_bookmark176) 173
7. [Learning with Uncertainty](#_bookmark178) 177
   1. [K-means](#_bookmark180) 177

[10.2 EM](#_bookmark181) 181

1. [Multiagent Systems](#_bookmark184) 185
   1. [Minimax](#_bookmark186) 185

Contents 5

1. [Reinforcement Learning](#_bookmark192) 191
   1. [Representing Agents and Environments](#_bookmark194) 191
   2. [Q Learning](#_bookmark199) 197
   3. [Model-based Reinforcement Learner](#_bookmark203) 200
   4. [Reinforcement Learning with Features](#_bookmark205) 202
   5. [Learning to coordinate - UNFINISHED!!!!](#_bookmark209) 208
2. [Relational Learning](#_bookmark210) 209
   1. [Collaborative Filtering](#_bookmark212) 209

[Index](#_bookmark215) 217

Chapter 1

Python for Artificial Intelligence

## Why Python?

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

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

## Getting Python

You need Python 3 (<http://python.org/>) and matplotlib ([http://matplotlib.](http://matplotlib.org/) [org/](http://matplotlib.org/)) that runs with Python 3. This code is *not* compatible with Python 2 (e.g., with Python 2.7).

Download and istall the latest Python 3 release from <http://python.org/>.

This should also install *pip*3. You can install matplotlib using

pip3 install matplotlib

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

pip instead of pip3.

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

7

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

pip3 install --upgrade matplotlib

We recommend using the enhanced interactive python **ipython** ([http://](http://ipython.org/) [ipython.org/](http://ipython.org/)). To install ipython after you have installed python do:

pip3 install ipython

## Running Python

We assume that everything is done with an interactive Python shell. You can either do this with an IDE, such as IDLE that comes with standard Python distributions, or just running ipython3 (or perhaps just ipython) from a shell.

Here we describe the most simple version that uses no IDE. If you down- load the zip file, and cd to the “aipython” folder where the .py files are, you should be able to do the following, with user input following : . The first ipython3 command is in the operating system shell (note that the -i is impor- tant to enter interactive mode):

$ ipython3 -i searchGeneric.py

Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)

Type 'copyright', 'credits' or 'license' for more information IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help. Testing problem 1:

7 paths have been expanded and 4 paths remain in the frontier Path found: a --> b --> c --> d --> g

Passed unit test

In [1]: searcher2 = AStarSearcher(searchProblem.acyclic\_delivery\_problem) #A\* In [2]: searcher2.search() # find first path

16 paths have been expanded and 5 paths remain in the frontier Out[2]: o103 --> o109 --> o119 --> o123 --> r123

In [3]: searcher2.search() # find next path

21 paths have been expanded and 6 paths remain in the frontier Out[3]: o103 --> b3 --> b4 --> o109 --> o119 --> o123 --> r123

In [4]: searcher2.search() # find next path

28 paths have been expanded and 5 paths remain in the frontier

Out[4]: o103 --> b3 --> b1 --> b2 --> b4 --> o109 --> o119 --> o123 --> r123

In [5]: searcher2.search() # find next path

No (more) solutions. Total of 33 paths expanded.

##### Pitfalls 9

In [6]:

You can then interact at the last prompt.

There are many textbooks for Python. The best source of information about python is <https://www.python.org/>. We will be using Python 3; please down- load the latest release. The documentation is at <https://docs.python.org/3/>. The rest of this chapter is about what is special about the code for AI tools.

We will only use the Standard Python Library and matplotlib. All of the exer- cises 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.

## Pitfalls

It is important to know when side effects occur. Often AI programs consider what would happen or what may have happened. In many such cases, we don’t want side effects. When an agent acts in the world, side effects are ap- propriate.

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 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.

## Features of Python

* + 1. Lists, Tuples, Sets, Dictionaries and Comprehensions

We make extensive uses of lists, tuples, sets and dictionaries (dicts). See

<https://docs.python.org/3/library/stdtypes.html>

One of the nice features of Python is the use of list comprehensions (and also tuple, set and dictionary comprehensions).

(*fe* for *e* in *iter* if *cond*)

#### enumerates the values *fe* for each *e* in *iter* for which *cond* is true. The “if cond” part is optional, but the “for” and “in” are not optional. Here *e* has to be a variable, *iter* is an iterator, which can generate a stream of data, such as a list, a set, a range object (to enumerate integers between ranges) or a file. *cond*

is an expression that evaluates to either True or False for each *e*, and *fe* is an expression that will be evaluated for each value of *e* for which *cond* returns *True*.

The result can go in a list or used in another iteration, or can be called directly using *next*. The procedure *next* takes an iterator returns the next el- ement (advancing the iterator) and raises a StopIteration exception if there is no next element. The following shows a simple example, where user input is prepended with >>>

>>> [e\*e for e in range(20) if e%2==0]

[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]

>>> a = (e\*e for e in range(20) if e%2==0)

>>> next(a) 0

>>> next(a) 4

>>> next(a) 16

>>> list(a)

[36, 64, 100, 144, 196, 256, 324]

>>> next(a)

Traceback (most recent call last): File "<stdin>", line 1, in <module>

StopIteration

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

Comprehensions can also be used for dictionaries. The following code cre-

ates an index for list *a*:

>>> a = ["a","f","bar","b","a","aaaaa"]

>>> ind = {a[i]:i for i in range(len(a))}

>>> ind

{'a': 4, 'f': 1, 'bar': 2, 'b': 3, 'aaaaa': 5}

>>> ind['b'] 3

which means that 'b' is the 3rd element of the list.

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

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

where *enumerate* returns an iterator of (*index*, *value*) pairs.

### 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

##### Features of Python 11

*called*, not the value of the variable when the function was defined (this is called “late binding”). This means if you want to use the value a variable has when the function is created, you need to save the current value of that variable. Whereas Python uses “late binding” by default, the alternative that newcomers often expect is “early binding”, where a function uses the value a variable had when the function was defined, can be easily implemented.

Consider the following programs designed to create a list of 5 functions, where the *i*th function in the list is meant to add *i* to its argument:[1](#_bookmark11)

pythonDemo.py — Some tricky examples

11 fun\_list1 = []

12 **for** i **in range**(5):

13 **def** fun1(e):

14 **return** e+i

15 fun\_list1.append(fun1)

16

17 fun\_list2 = []

18 **for** i **in range**(5):

19 **def** fun2(e,iv=i):

20 **return** e+iv

21 fun\_list2.append(fun2)

22

23 fun\_list3 = [**lambda** e: e+i **for** i **in range**(5)]

24

25 fun\_list4 = [**lambda** e,iv=i: e+iv **for** i **in range**(5)]

26

27 i=56

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

pythonDemo.py — (continued)

29 # in Shell do

30 ## ipython -i pythonDemo.py

31 # Try these (copy text after the comment symbol and paste in the Python prompt):

32 # print([f(10) for f in fun\_list1])

33 # print([f(10) for f in fun\_list2])

34 # print([f(10) for f in fun\_list3])

35 # print([f(10) for f in fun\_list4])

#### In the first for-loop, the function *fun* uses *i*, whose value is the last value it was assigned. In the second loop, the function *fun*2 uses *iv*. There is a separate *iv* variable for each function, and its value is the value of *i* when the function was defined. Thus *fun*1 uses late binding, and *fun*2 uses early binding. *fun list*3 and *fun list*4 are equivalent to the first two (except *fun list*4 uses a different *i* variable).

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

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

* + 1. Generators and Coroutines

Python has generators which can be used for a form of coroutines.

The *yield* command returns a value that is obtained with *next*. It is typically used to enumerate the values for a *for* loop or in generators.

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

pythonDemo.py — (continued)

37 **def** myrange(start, stop, step=1):

38 """enumerates the values from start in steps of size step that are

39 less than stop.

40 """

41 **assert** step>0, "only positive steps implemented in myrange"

42 i = start

43 **while** i<stop:

44 yield i

45 i += step

46

47 **print**("myrange(2,30,3):",**list**(myrange(2,30,3)))

#### Note that the built-in *range* is unconventional in how it handles a single ar- gument, as the single argument acts as the second argument of the function. Note also that the built-in range also allows for indexing (e.g., *range*(2, 30, 3)[2] returns 8), which the above implementation does not. However *myrange* also works for floats, which the built-in range does not.

**Exercise 1.1** Implement a version of *myrange* that acts like the built-in version when there is a single argument. (Hint: make the second argument have a default value that can be recognized in the function.)

Yield can be used to generate the same sequence of values as in the example of Section [1.5.1:](#_bookmark9)

pythonDemo.py — (continued)

49 **def** ga(n):

50 """generates square of even nonnegative integers less than n"""

51 **for** e **in range**(n):

52 **if** e%2==0:

53 yield e\*e

54 a = ga(20)

#### The sequence of next(a), and list(a) gives exactly the same results as the com- prehension in Section [1.5.1.](#_bookmark9)

##### Useful Libraries 13

It is straightforward to write a version of the built-in *enumerate*. Let’s call it

*myenumerate*:

pythonDemo.py — (continued)

56 **def** myenumerate(enum):

57 **for** i **in range**(**len**(enum)):

58 yield i,enum[i]

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

## Useful Libraries

* + 1. Timing Code

In order to compare algorithms, we often want to compute how long a program takes; this is called the **runtime** of the program. The most straightforward way to compute runtime is to use *time*.*perf counter*(), as in:

import time

start\_time = time.perf\_counter() compute\_for\_a\_while()

end\_time = time.perf\_counter()

print("Time:", end\_time - start\_time, "seconds")

If this time is very small (say less than 0.2 second), it is probably very inac- curate, and it may be better to run your code many times to get a more accu- rate count. For this you can use *timeit* ([https://docs.python.org/3/library/](https://docs.python.org/3/library/timeit.html) [timeit.html](https://docs.python.org/3/library/timeit.html)). To use timeit to time the call to *foo*.*bar*(*aaa*) use:

import timeit

time = timeit.timeit("foo.bar(aaa)",

setup="from main import foo,aaa", number=100)

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

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

* + 1. Plotting: Matplotlib

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

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

pythonDemo.py — (continued)

60 **import** matplotlib.pyplot as plt

61

62 **def** myplot(**min**,**max**,step,fun1,fun2):

63 plt.ion() # make it interactive

64 plt.xlabel("The x axis")

65 plt.ylabel("The y axis")

66 plt.xscale('linear') # Makes a 'log' or 'linear' scale

67 xvalues = **range**(**min**,**max**,step)

68 plt.plot(xvalues,[fun1(x) **for** x **in** xvalues],

69 label="The first fun")

70 plt.plot(xvalues,[fun2(x) **for** x **in** xvalues], linestyle='--',color='k',

71 label=fun2. doc ) # use the doc string of the function

72 plt.legend(loc="upper right") # display the legend

73

74 **def** slin(x):

75 """y=2x+7"""

76 **return** 2\*x+7

77 **def** sqfun(x):

78 """y=(x-40)ˆ2/10-20"""

79 **return** (x-40)\*\*2/10-20

80

81 # Try the following:

82 # from pythonDemo import myplot, slin, sqfun

83 # import matplotlib.pyplot as plt

84 # myplot(0,100,1,slin,sqfun)

85 # plt.legend(loc="best")

86 # import math

87 # plt.plot([41+40\*math.cos(th/10) for th in range(50)],

88 # [100+100\*math.sin(th/10) for th in range(50)])

89 # plt.text(40,100,"ellipse?")

90 # plt.xscale('log')

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

## Utilities

* + 1. Display

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

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

*self* .*display*(*level*, *to print* . . . )

##### Utilities 15

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

16

17 **def** display(self,level,\*args,\*\*nargs):

18 """print the arguments if level is less than or equal to the

19 current max\_display\_level.

20 level is an integer.

21 the other arguments are whatever arguments print can take.

22 """

23 **if** level <= self.max\_display\_level:

24 **print**(\*args, \*\*nargs) ##if error you are using Python2 not Python3

#### Note that *args* gets a tuple of the positional arguments, and *nargs* gets a dictio- nary of the keyword arguments). This will not work in Python 2, and will give an error.

Any class that wants to use *display* can be made a subclass of *Displayable*. To change the maximum display level to say 3, for a class do:

##### Classname.max display level = 3

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

The value of *max display level* by convention is:

#### display nothing

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

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

display.py — (continued)

26 **def** visualize(func):

27 """A decorator for algorithms that do interactive visualization.

28 Ignored here.

29 """

30 **return** func

### Argmax

#### Python has a built-in *max* function that takes a generator (or a list or set) and re- turns the maximum value. The *argmax* method returns the index of an element that has the maximum value. If there are multiple elements with the maximum value, one if the indexes to that value is returned at random. This assumes a generator of (*element*, *value*) pairs, as for example is generated by the built-in *enumerate*.

utilities.py — AIPython useful utilities

11 **import** random

12

13 **def** argmax(gen):

14 """gen is a generator of (element,value) pairs, where value is a real.

15 argmax returns an element with maximal value.

16 If there are multiple elements with the max value, one is returned at random.

17 """

18 maxv = **float**('-Infinity') # 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** random.choice(maxvals)

26

27 # Try:

28 # argmax(enumerate([1,6,3,77,3,55,23]))

**Exercise 1.3** Change argmax to have an optinal argument that specifies whether you want the “first”, “last” or a “random” index of the maximum value returned. If you want the first or the last, you don’t need to keep a list of the maximum elements.

* + 1. Probability

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

utilities.py — (continued)

30 **def** flip(prob):

31 """return true with probability prob"""

32 **return** random.random() < prob

* 1. *Testing Code* 17
     1. Dictionary Union

The function *dict union*(*d*1, *d*2) returns the union of dictionaries *d*1 and *d*2. If the values for the keys conflict, the values in *d*2 are used. This is similar to *dict*(*d*1, ∗ ∗ *d*2), but that only works when the keys of *d*2 are strings.

utilities.py — (continued)

34 **def** dict\_union(d1,d2):

35 """returns a dictionary that contains the keys of d1 and d2.

36 The value for each key that is in d2 is the value from d2,

37 otherwise it is the value from d1.

38 This does not have side effects.

39 """

40 d = **dict**(d1) # copy d1

41 d.update(d2)

42 **return** d

## Testing Code

It is important to test code early and test it often. We include a simple form of **unit tests**. The value of the current module is in name and if the module is run at the top-level, it’s value is " main ". See [https://docs.python.org/3/](https://docs.python.org/3/library/__main__.html) [library/ main .html](https://docs.python.org/3/library/__main__.html).

The following code tests argmax and dict\_union, but only when if utilities is loaded in the top-level. If it is loaded in a module the test code is not run.

In your code you should do more substantial testing than we do here, in particular testing the boundary cases.

utilities.py — (continued)

44 **def** test():

45 """Test part of utilities"""

46 **assert** argmax(**enumerate**([1,6,55,3,55,23])) **in** [2,4]

47 **assert** dict\_union({1:4, 2:5, 3:4},{5:7, 2:9}) == {1:4, 2:9, 3:4, 5:7}

48 **print**("Passed unit test in utilities")

49

50 **if** name == " main ":

51 test()

# Chapter 2

Agents and Control

#### This implements the controllers described in Chapter 2.

In this version the higher-levels call the lower-levels. A more sophisti- cated version may have them run concurrently (either as coroutines or in paral- lel). The higher-levels calling the lower-level works in simulated environments when there is a single agent, and where the lower-level are written to make sure they return (and don’t go on forever), and the higher level doesn’t take too long (as the lower-levels will wait until called again).

## Representing Agents and Environments

An agent observes the world, and carries out actions in the environment, it also has an internal state that it updates. The environment takes in actions of the agents, updates it internal state and returns the percepts.

In this implementation, the state of the agent and the state of the environ- ment are represented using standard Python variables, which are updated as the state changes. The percepts and the actions are represented as variable- value dictionaries.

An agent implements the *go*(*n*) method, where *n* is an integer. This means that the agent should run for *n* time steps.

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

agents.py — Agent and Controllers

11 **import** random

12

13 **class** Agent(**object**):

14 **def** init (self,env):

#### 19

15 """set up the agent"""

16 self.env=env

17

18 **def** go(self,n):

19 """acts for n time steps"""

20 **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 dictio- nary. The use of dictionaries allows for structured actions and percepts.

Note that *Environment* is a subclass of *Displayable* so that it can use the

#### *display* method described in Section [1.7.1.](#_bookmark16)

agents.py — (continued)

22 **from** display **import** Displayable

23 **class** Environment(Displayable):

24 **def** initial\_percepts(self):

25 """returns the initial percepts for the agent"""

26 **raise** NotImplementedError("initial\_percepts") # abstract method

27

28 **def** do(self,action):

29 """does the action in the environment

30 returns the next percept """

31 **raise** NotImplementedError("do") # abstract method

## Paper buying agent and environment

To run the demo, in folder ”aipython”, load ”agents.py”, using e.g., ipython -i agents.py, and copy and paste the commented-out commands at the bottom of that file. This requires Python 3 with matplotlib.

#### This is an implementation of the paper buying example.

* + 1. The Environment

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

Here we assume that the prices are obtained from the *prices* list plus a ran- dom integer in range [0, *max price addon*) plus a linear ”inflation”. The agent cannot access the price model; it just observes the prices and the amount in stock.

agents.py — (continued)

33 **class** TP\_env(Environment):

##### 2.2. Paper buying agent and environment 21

34 prices = [234, 234, 234, 234, 255, 255, 275, 275, 211, 211, 211,

35 234, 234, 234, 234, 199, 199, 275, 275, 234, 234, 234, 234, 255,

36 255, 260, 260, 265, 265, 265, 265, 270, 270, 255, 255, 260, 260,

37 265, 265, 150, 150, 265, 265, 270, 270, 255, 255, 260, 260, 265,

38 265, 265, 265, 270, 270, 211, 211, 255, 255, 260, 260, 265, 265,

39 260, 265, 270, 270, 205, 255, 255, 260, 260, 265, 265, 265, 265,

40 270, 270]

41 max\_price\_addon = 20 # maximum of random value added to get price

42

43 **def** init (self):

44 """paper buying agent"""

45 self.time=0

46 self.stock=20

47 self.stock\_history = [] # memory of the stock history

48 self.price\_history = [] # memory of the price history

49

50 **def** initial\_percepts(self):

51 """return initial percepts"""

52 self.stock\_history.append(self.stock)

53 price = self.prices[0]+random.randrange(self.max\_price\_addon)

54 self.price\_history.append(price)

55 **return** {'price': price,

56 'instock': self.stock}

57

58 **def** do(self, action):

59 """does action (buy) and returns percepts (price and instock)"""

60 used = pick\_from\_dist({6:0.1, 5:0.1, 4:0.2, 3:0.3, 2:0.2, 1:0.1})

61 bought = action['buy']

62 self.stock = self.stock+bought-used

63 self.stock\_history.append(self.stock)

64 self.time += 1

65 price = (self.prices[self.time%**len**(self.prices)] # repeating pattern

66 +random.randrange(self.max\_price\_addon) # plus randomness

67 +self.time//2) # plus inflation

68 self.price\_history.append(price)

69 **return** {'price': price,

70 'instock': self.stock}

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

agents.py — (continued)

72 **def** pick\_from\_dist(item\_prob\_dist):

73 """ returns a value from a distribution.

74 item\_prob\_dist is an item:probability dictionary, where the

75 probabilities sum to 1.

76 returns an item chosen in proportion to its probability

77 """

78 ranreal = random.random()

79 **for** (it,prob) **in** item\_prob\_dist.items():

80 **if** ranreal < prob:

81 **return** it

82 **else**:

83 ranreal -= prob

84 **raise** RuntimeError(**str**(item\_prob\_dist)+" is not a probability distribution")

* + 1. The Agent

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

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

agents.py — (continued)

86 **class** TP\_agent(Agent):

87 **def** init (self, env):

88 self.env = env

89 self.spent = 0

90 percepts = env.initial\_percepts()

91 self.ave = self.last\_price = percepts['price']

92 self.instock = percepts['instock']

93

94 **def** go(self, n):

95 """go for n time steps

96 """

97 **for** i **in range**(n):

98 **if** self.last\_price < 0.9\*self.ave **and** self.instock < 60:

99 tobuy = 48

100 **elif** self.instock < 12:

101 tobuy = 12

102 **else**:

103 tobuy = 0

104 self.spent += tobuy\*self.last\_price

105 percepts = env.do({'buy': tobuy})

106 self.last\_price = percepts['price']

107 self.ave = self.ave+(self.last\_price-self.ave)\*0.05

108 self.instock = percepts['instock']

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

agents.py — (continued)

110 env = TP\_env()

111 ag = TP\_agent(env)

112 #ag.go(90)

113 #ag.spent/env.time ## average spent per time period

### Plotting

#### The following plots the price and number in stock history:

|  |  |  |
| --- | --- | --- |
|  |  | agents.py — (continued) |
| 115 |  | **import** matplotlib.pyplot as plt |
| 116 |  |  |
| 117 |  | **class** Plot\_prices(**object**): |
| 118 |  | """Set up the plot for history of price and number in stock""" |
| 119 |  | **def** init (self, ag,env): |

120 self.ag = ag

121 self.env = env

122 plt.ion()

123 plt.xlabel("Time")

124 plt.ylabel("Number in stock. Price.")

125

126 **def** plot\_run(self):

127 """plot history of price and instock"""

128 num = **len**(env.stock\_history)

129 plt.plot(**range**(num),env.stock\_history,label="In stock")

130 plt.plot(**range**(num),env.price\_history,label="Price")

131 #plt.legend(loc="upper left")

132 plt.draw()

133

134 # pl = Plot\_prices(ag,env)

135 # ag.go(90); pl.plot\_run()

## Hierarchical Controller

To run the hierarchical controller, in folder ”aipython”, load ”agentTop.py”, using e.g., ipython -i agentTop.py, and copy and paste the commands near the bottom of that file. This requires Python 3 with matplotlib.

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

We arbitrarily divide the environment and the body, so that the environ- ment just defines the walls, and the body includes everything to do with the agent. Note that the named locations are part of the (top-level of the) agent, not part of the environment, although they could have been.

* + 1. Environment

The environment defines the walls.

agentEnv.py — Agent environment

11 **import** math

12 **from** agents **import** Environment

13

14 **class** Rob\_env(Environment):

15 **def** init (self,walls = {}):

16 """walls is a set of line segments

17 where each line segment is of the form ((x0,y0),(x1,y1))

18 """

19 self.walls = walls

* + 1. Body

#### The body defines everything about the agent body.

agentEnv.py — (continued)

21 **import** math

22 **from** agents **import** Environment

23 **import** matplotlib.pyplot as plt

24 **import** time

25

26 **class** Rob\_body(Environment):

27 **def** init (self, env, init\_pos=(0,0,90)):

28 """ env is the current environment

29 init\_pos is a triple of (x-position, y-position, direction)

30 direction is in degrees; 0 is to right, 90 is straight-up, etc

31 """

32 self.env = env

33 self.rob\_x, self.rob\_y, self.rob\_dir = init\_pos

34 self.turning\_angle = 18 # degrees that a left makes

35 self.whisker\_length = 6 # length of the whisker

36 self.whisker\_angle = 30 # angle of whisker relative to robot

37 self.crashed = False

38 # The following control how it is plotted

39 self.plotting = True # whether the trace is being plotted

40 self.sleep\_time = 0.05 # time between actions (for real-time plotting)

41 # The following are data structures maintained:

42 self.history = [(self.rob\_x, self.rob\_y)] # history of (x,y) positions

43 self.wall\_history = [] # history of hitting the wall

44

45 **def** percepts(self):

46 **return** {'rob\_x\_pos':self.rob\_x, 'rob\_y\_pos':self.rob\_y,

47 'rob\_dir':self.rob\_dir, 'whisker':self.whisker() , 'crashed':self.crashed}

48 initial\_percepts = percepts # use percept function for initial percepts too

49

50 **def** do(self,action):

51 """ action is {'steer':direction}

52 direction is 'left', 'right' or 'straight'

53 """

54 **if** self.crashed:

55 **return** self.percepts()

56 direction = action['steer']

57 compass\_deriv = {'left':1,'straight':0,'right':-1}[direction]\*self.turning\_angle

58 self.rob\_dir = (self.rob\_dir + compass\_deriv +360)%360 # make in range [0,360)

59 rob\_x\_new = self.rob\_x + math.cos(self.rob\_dir\*math.pi/180)

60 rob\_y\_new = self.rob\_y + math.sin(self.rob\_dir\*math.pi/180)

61 path = ((self.rob\_x,self.rob\_y),(rob\_x\_new,rob\_y\_new))

62 **if any**(line\_segments\_intersect(path,wall) **for** wall **in** self.env.walls):

63 self.crashed = True

64 **if** self.plotting:

65 plt.plot([self.rob\_x],[self.rob\_y],"r\*",markersize=20.0)

66 plt.draw()

67 self.rob\_x, self.rob\_y = rob\_x\_new, rob\_y\_new

68 self.history.append((self.rob\_x, self.rob\_y))

69 **if** self.plotting **and not** self.crashed:

70 plt.plot([self.rob\_x],[self.rob\_y],"go")

71 plt.draw()

72 plt.pause(self.sleep\_time)

73 **return** self.percepts()

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

agentEnv.py — (continued)

75 **def** whisker(self):

76 """returns true whenever the whisker sensor intersects with a wall

77 """

78 whisk\_ang\_world = (self.rob\_dir-self.whisker\_angle)\*math.pi/180

79 # angle in radians in world coordinates

80 wx = self.rob\_x + self.whisker\_length \* math.cos(whisk\_ang\_world)

81 wy = self.rob\_y + self.whisker\_length \* math.sin(whisk\_ang\_world)

82 whisker\_line = ((self.rob\_x,self.rob\_y),(wx,wy))

83 hit = **any**(line\_segments\_intersect(whisker\_line,wall)

84 **for** wall **in** self.env.walls)

85 **if** hit:

86 self.wall\_history.append((self.rob\_x, self.rob\_y))

87 **if** self.plotting:

88 plt.plot([self.rob\_x],[self.rob\_y],"ro")

89 plt.draw()

90 **return** hit

91

92 **def** line\_segments\_intersect(linea,lineb):

93 """returns true if the line segments, linea and lineb intersect.

94 A line segment is represented as a pair of points.

95 A point is represented as a (x,y) pair.

96 """

97 ((x0a,y0a),(x1a,y1a)) = linea

98 ((x0b,y0b),(x1b,y1b)) = lineb

99 da, db = x1a-x0a, x1b-x0b

100 ea, eb = y1a-y0a, y1b-y0b

101 denom = db\*ea-eb\*da

102 **if** denom==0: # line segments are parallel

103 **return** False

104 cb = (da\*(y0b-y0a)-ea\*(x0b-x0a))/denom # position along line b

105 **if** cb<0 **or** cb>1:

106 **return** False

|  |  |
| --- | --- |
| 107 | ca = (db\*(y0b-y0a)-eb\*(x0b-x0a))/denom # position along line a |
| 108 | **return** 0<=ca<=1 |
| 109 |  |
| 110 | # Test cases: |
| 111 | # assert line\_segments\_intersect(((0,0),(1,1)),((1,0),(0,1))) |
| 112 | # assert not line\_segments\_intersect(((0,0),(1,1)),((1,0),(0.6,0.4))) |
| 113 | # assert line\_segments\_intersect(((0,0),(1,1)),((1,0),(0.4,0.6))) |

### Middle Layer

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

agentMiddle.py — Middle Layer

11 **from** agents **import** Environment

12 **import** math

13

14 **class** Rob\_middle\_layer(Environment):

15 **def** init (self,env):

16 self.env=env

17 self.percepts = env.initial\_percepts()

18 self.straight\_angle = 11 # angle that is close enough to straight ahead

19 self.close\_threshold = 2 # distance that is close enough to arrived

20 self.close\_threshold\_squared = self.close\_threshold\*\*2 # just compute it once

21

22 **def** initial\_percepts(self):

23 **return** {}

24

25 **def** do(self, action):

26 """action is {'go\_to':target\_pos,'timeout':timeout}

27 target\_pos is (x,y) pair

28 timeout is the number of steps to try

29 returns {'arrived':True} when arrived is true

30 or {'arrived':False} if it reached the timeout

31 """

32 **if** 'timeout' **in** action:

33 remaining = action['timeout']

34 **else**:

35 remaining = -1 # will never reach 0

36 target\_pos = action['go\_to']

37 arrived = self.close\_enough(target\_pos)

38 **while not** arrived **and** remaining != 0:

39 self.percepts = self.env.do({"steer":self.steer(target\_pos)})

40 remaining -= 1

41 arrived = self.close\_enough(target\_pos)

42 **return** {'arrived':arrived}

#### This determines how to steer depending on whether the goal is to the right or the left of where the robot is facing.

agentMiddle.py — (continued)

44 **def** steer(self,target\_pos):

45 **if** self.percepts['whisker']:

46 self.display(3,'whisker on', self.percepts)

47 **return** "left"

48 **else**:

49 gx,gy = target\_pos

50 rx,ry = self.percepts['rob\_x\_pos'],self.percepts['rob\_y\_pos']

51 goal\_dir = math.acos((gx-rx)/math.sqrt((gx-rx)\*(gx-rx)

52 +(gy-ry)\*(gy-ry)))\*180/math.pi

53 **if** ry>gy:

54 goal\_dir = -goal\_dir

55 goal\_from\_rob = (goal\_dir - self.percepts['rob\_dir']+540)%360-180

56 **assert** -180 < goal\_from\_rob <= 180

57 **if** goal\_from\_rob > self.straight\_angle:

58 **return** "left"

59 **elif** goal\_from\_rob < -self.straight\_angle:

60 **return** "right"

61 **else**:

62 **return** "straight"

63

64 **def** close\_enough(self,target\_pos):

65 gx,gy = target\_pos

66 rx,ry = self.percepts['rob\_x\_pos'],self.percepts['rob\_y\_pos']

67 **return** (gx-rx)\*\*2 + (gy-ry)\*\*2 <= self.close\_threshold\_squared

### Top Layer

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

agentTop.py — Top Layer

11 **from** agentMiddle **import** Rob\_middle\_layer

12 **from** agents **import** Environment

13

14 **class** Rob\_top\_layer(Environment):

15 **def** init (self, middle, timeout=200, locations = {'mail':(-5,10),

16 'o103':(50,10), 'o109':(100,10),'storage':(101,51)} ):

17 """middle is the middle layer

18 timeout is the number of steps the middle layer goes before giving up

19 locations is a loc:pos dictionary

20 where loc is a named location, and pos is an (x,y) position.

21 """

22 self.middle = middle

23 self.timeout = timeout # number of steps before the middle layer should give up

24 self.locations = locations

25

26 **def** do(self,plan):

27 """carry out actions.

28 actions is of the form {'visit':list\_of\_locations}

29 It visits the locations in turn.

30 """

31 to\_do = plan['visit']

32 **for** loc **in** to\_do:

33 position = self.locations[loc]

34 arrived = self.middle.do({'go\_to':position, 'timeout':self.timeout})

35 self.display(1,"Arrived at",loc,arrived)

### Plotting

#### The following is used to plot the locations, the walls and (eventually) the move- ment of the robot. It can either plot the movement if the robot as it is go- ing (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*()).

|  |  |  |  |
| --- | --- | --- | --- |
| 37  38  39  40  41 |  | **import class** P  **def** | agentTop.py — (continued)  matplotlib.pyplot as plt  lot\_env(**object**):  init (self, body,top): """sets up the plot |
| 42 |  |  | """ |
| 43 |  |  | self.body = body |
| 44 |  |  | plt.ion() |
| 45 |  |  | plt.clf() |
| 46 |  |  | plt.axes().set\_aspect('equal') |
| 47 |  |  | **for** wall **in** body.env.walls: |
| 48 |  |  | ((x0,y0),(x1,y1)) = wall |
| 49 |  |  | plt.plot([x0,x1],[y0,y1],"-k",linewidth=3) |
| 50 |  |  | **for** loc **in** top.locations: |
| 51 |  |  | (x,y) = top.locations[loc] |
| 52 |  |  | plt.plot([x],[y],"k<") |
| 53 |  |  | plt.text(x+1.0,y+0.5,loc) # print the label above and to the right |
| 54 |  |  | plt.plot([body.rob\_x],[body.rob\_y],"go") |
| 55 |  |  | plt.draw() |
| 56 |  |  |  |
| 57 |  | **def** | plot\_run(self): |
| 58 |  |  | """plots the history after the agent has finished. |
| 59 |  |  | This is typically only used if body.plotting==False |
| 60 |  |  | """ |
| 61 |  |  | xs,ys = **zip**(\*self.body.history) |
| 62 |  |  | plt.plot(xs,ys,"go") |
| 63 |  |  | wxs,wys = **zip**(\*self.body.wall\_history) |
| 64 |  |  | plt.plot(wxs,wys,"ro") |
| 65 |  |  | #plt.draw() |

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

agentTop.py — (continued)

67 **from** agentEnv **import** Rob\_body, Rob\_env

68

69 env = Rob\_env({((20,0),(30,20)), ((70,-5),(70,25))})

70 body = Rob\_body(env)

71 middle = Rob\_middle\_layer(body)

72 top = Rob\_top\_layer(middle)

73

74 # try:

75 # pl=Plot\_env(body,top)

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

77 # You can directly control the middle layer:

78 # middle.do({'go\_to':(30,-10), 'timeout':200})

79 # Can you make it crash?

**Exercise 2.1** The following code implements a robot trap. Write a controller that can escape the “trap” and get to the goal. See textbook for hints.

agentTop.py — (continued)

81 # Robot Trap for which the current controller cannot escape:

82 trap\_env = Rob\_env({((10,-21),(10,0)), ((10,10),(10,31)), ((30,-10),(30,0)),

83 ((30,10),(30,20)), ((50,-21),(50,31)), ((10,-21),(50,-21)),

84 ((10,0),(30,0)), ((10,10),(30,10)), ((10,31),(50,31))})

85 trap\_body = Rob\_body(trap\_env,init\_pos=(-1,0,90))

86 trap\_middle = Rob\_middle\_layer(trap\_body)

87 trap\_top = Rob\_top\_layer(trap\_middle,locations={'goal':(71,0)})

88

89 # Robot trap exercise:

90 # pl=Plot\_env(trap\_body,trap\_top)

91 # trap\_top.do({'visit':['goal']})

# Chapter 3

Searching for Solutions

## Representing Search Problems

#### A search problem consists of:

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

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

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

searchProblem.py — representations of search problems

11 **class** Search\_problem(**object**):

12 """A search problem consists of:

13 \* a start node

14 \* a neighbors function that gives the neighbors of a node

15 \* a specification of a goal

16 \* a (optional) heuristic function.

#### 31

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

18

19 **def** start\_node(self):

20 """returns start node"""

21 **raise** NotImplementedError("start\_node") # abstract method

22

23 **def** is\_goal(self,node):

24 """is True if node is a goal"""

25 **raise** NotImplementedError("is\_goal") # abstract method

26

27 **def** neighbors(self,node):

28 """returns a list of the arcs for the neighbors of node"""

29 **raise** NotImplementedError("neighbors") # abstract method

30

31 **def** heuristic(self,n):

32 """Gives the heuristic value of node n.

33 Returns 0 if not overridden."""

34 **return** 0

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

( )

searchProblem.py — (continued)

36 **class** Arc(**object**):

37 """An arc has a from\_node and a to\_node node and a (non-negative) cost"""

38 **def** init (self, from\_node, to\_node, cost=1, action=None):

39 **assert** cost >= 0, ("Cost cannot be negative for"+

40 **str**(from\_node)+"->"+**str**(to\_node)+", cost: "+**str**(cost))

41 self.from\_node = from\_node

42 self.to\_node = to\_node

43 self.action = action

44 self.cost=cost

45

46 **def** repr (self):

47 """string representation of an arc"""

48 **if** self.action:

49 **return str**(self.from\_node)+" --"+**str**(self.action)+"--> "+**str**(self.to\_node)

50 **else**:

51 **return str**(self.from\_node)+" --> "+**str**(self.to\_node)

* + 1. Explicit Representation of Search Graph

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

An **explicit graph** consists of

#### a list or set of nodes

* + - * a list or set of arcs
        + a start node
        + a list or set of goal nodes
        + (optionally) a dictionary that maps a node to a heuristic value for that node

To define a search problem, we need to define the start node, the goal predicate, the neighbors function and the heuristic function.

searchProblem.py — (continued)

53 **class** Search\_problem\_from\_explicit\_graph(Search\_problem):

54 """A search problem consists of:

|  |  |  |
| --- | --- | --- |
| 55 | \* a | list or set of nodes |
| 56 | \* a | list or set of arcs |
| 57 | \* a | start node |
| 58 | \* a | list or set of goal nodes |
| 59 | \* a | dictionary that maps each node into its heuristic value. |
| 60 | """ |  |
| 61 |  |  |
| 62 | **def** | init (self, nodes, arcs, start=None, goals=**set**(), hmap={}): |
| 63 |  | self.neighs = {} |
| 64 |  | self.nodes = nodes |
| 65 |  | **for** node **in** nodes: |
| 66 |  | self.neighs[node]=[] |
| 67 |  | self.arcs = arcs |
| 68  69 |  | **for** arc **in** arcs: self.neighs[arc.from\_node].append(arc) |
| 70 |  | self.start = start |
| 71 |  | self.goals = goals |
| 72 |  | self.hmap = hmap |
| 73 |  |  |
| 74 | **def** | start\_node(self): |
| 75 |  | """returns start node""" |
| 76 |  | **return** self.start |
| 77 |  |  |
| 78 | **def** | is\_goal(self,node): |
| 79 |  | """is True if node is a goal""" |
| 80 |  | **return** node **in** self.goals |
| 81 |  |  |
| 82 | **def** | neighbors(self,node): |
| 83 |  | """returns the neighbors of node""" |
| 84 |  | **return** self.neighs[node] |
| 85 |  |  |
| 86 | **def** | heuristic(self,node): |
| 87 |  | """Gives the heuristic value of node n. |
| 88 |  | Returns 0 if not overridden in the hmap.""" |
| 89 |  | **if** node **in** self.hmap: |
| 90 |  | **return** self.hmap[node] |
| 91 |  | **else**: |
| 92 |  | **return** 0 |

93

94 **def** repr (self):

95 """returns a string representation of the search problem"""

96 res=""

97 **for** arc **in** self.arcs:

98 res += **str**(arc)+". "

99 **return** res

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

searchProblem.py — (continued)

101 **def** neighbor\_nodes(self,node):

102 """returns an iterator over the neighbors of node"""

103 **return** (path.to\_node **for** path **in** self.neighs[node])

### Paths

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

A path is either:

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

#### These cases are distinguished in the following code by having *arc* = *None* if the path has length 0, in which case *initial* is the node of the path.

searchProblem.py — (continued)

105 **class** Path(**object**):

106 """A path is either a node or a path followed by an arc"""

107

108 **def** init (self,initial,arc=None):

109 """initial is either a node (in which case arc is None) or

110 a path (in which case arc is an object of type Arc)"""

111 self.initial = initial

112 self.arc=arc

|  |  |
| --- | --- |
| 113 | **if** arc **is** None: |
| 114 | self.cost=0 |
| 115 | **else**: |
| 116 | self.cost = initial.cost+arc.cost |
| 117 |  |
| 118 | **def** end(self): |
| 119 | """returns the node at the end of the path""" |
| 120 | **if** self.arc **is** None: |

|  |  |  |
| --- | --- | --- |
| 121 |  | **return** self.initial |
| 122 |  | **else**: |
| 123 |  | **return** self.arc.to\_node |
| 124 |  |  |
| 125 | **def** | nodes(self): |
| 126 |  | """enumerates the nodes for the path. |
| 127 |  | This starts at the end and enumerates nodes in the path backwards.""" |
| 128 |  | current = self |
| 129 |  | **while** current.arc **is not** None: |
| 130  131 | yield current.arc.to\_node current = current.initial | |
| 132 |  | yield current.initial |
| 133 |  |  |
| 134 | **def** | initial\_nodes(self): |
| 135 |  | """enumerates the nodes for the path before the end node. |
| 136 |  | This starts at the end and enumerates nodes in the path backwards.""" |
| 137 |  | **if** self.arc **is not** None: |
| 138 | **for** nd **in** self.initial.nodes(): yield nd # could be "yield from" | |
| 139 |  |  |
| 140 | **def** | repr (self): |
| 141 |  | """returns a string representation of a path""" |
| 142 |  | **if** self.arc **is** None: |
| 143 |  | **return str**(self.initial) |
| 144 |  | **elif** self.arc.action: |

145 **return** (**str**(self.initial)+"\n --"+**str**(self.arc.action)

146 +"--> "+**str**(self.arc.to\_node))

147 **else**:

148 **return str**(self.initial)+" --> "+**str**(self.arc.to\_node)

### 3.1.3 Example Search Problems

#### The first search problem is one with 5 nodes where the least-cost path is one with many arcs. See Figure [3.1.](#_bookmark39) Note that this example is used for the unit tests, so the test (in searchGeneric) will need to be changed if this is changed.

searchProblem.py — (continued)

150 problem1 = Search\_problem\_from\_explicit\_graph(

|  |  |
| --- | --- |
| 151 | {'a','b','c','d','g'}, |
| 152 | [Arc('a','b',1), Arc('a','c',3), Arc('b','d',3), Arc('b','c',1), |
| 153 | Arc('c','d',1), Arc('c','g',3), Arc('d','g',1)], |
| 154 | start = 'a', |
| 155 | goals = {'g'}) |

#### The second search problem is one with 8 nodes where many paths do not lead to the goal. See Figure [3.2.](#_bookmark40)

searchProblem.py — (continued)

157 problem2 = Search\_problem\_from\_explicit\_graph(

158 {'a','b','c','d','e','g','h','j'},

159 [Arc('a','b',1), Arc('b','c',3), Arc('b','d',1), Arc('d','e',3),



a

3

1

b

1

3

c

3

1

d

1

g

Figure 3.1: problem1



a

1

3

h

b

1

1

j

d

3

1

3

g

c

e

Figure 3.2: problem2

|  |  |
| --- | --- |
| 160 | Arc('d','g',1), Arc('a','h',3), Arc('h','j',1)], |
| 161 | start = 'a', |
| 162 | goals = {'g'}) |

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

searchProblem.py — (continued)

164 problem3 = Search\_problem\_from\_explicit\_graph(

|  |  |
| --- | --- |
| 165 | {'a','b','c','d','e','g','h','j'}, |
| 166 | [], |
| 167 | start = 'g', |
| 168 | goals = {'k','g'}) |

#### The acyclic delivery problem is the delivery problem described in Example

3.4 and shown in Figure 3.2 of the textbook.

searchProblem.py — (continued)

170 acyclic\_delivery\_problem = Search\_problem\_from\_explicit\_graph(

171 {'mail','ts','o103','o109','o111','b1','b2','b3','b4','c1','c2','c3',

172 'o125','o123','o119','r123','storage'},

|  |  |
| --- | --- |
| 173 | [Arc('ts','mail',6), |
| 174 | Arc('o103','ts',8), |
| 175 | Arc('o103','b3',4), |
| 176 | Arc('o103','o109',12), |
| 177 | Arc('o109','o119',16), |
| 178 | Arc('o109','o111',4), |
| 179 | Arc('b1','c2',3), |
| 180 | Arc('b1','b2',6), |
| 181 | Arc('b2','b4',3), |
| 182 | Arc('b3','b1',4), |
| 183 | Arc('b3','b4',7), |
| 184 | Arc('b4','o109',7), |
| 185 | Arc('c1','c3',8), |
| 186 | Arc('c2','c3',6), |
| 187 | Arc('c2','c1',4), |
| 188 | Arc('o123','o125',4), |
| 189 | Arc('o123','r123',4), |
| 190 | Arc('o119','o123',9), |
| 191 | Arc('o119','storage',7)], |
| 192 | start = 'o103', |
| 193 | goals = {'r123'}, |
| 194 | hmap = { |
| 195 | 'mail' : 26, |
| 196 | 'ts' : 23, |
| 197 | 'o103' : 21, |
| 198 | 'o109' : 24, |
| 199 | 'o111' : 27, |
| 200 | 'o119' : 11, |
| 201 | 'o123' : 4, |
| 202 | 'o125' : 6, |
| 203 | 'r123' : 0, |
| 204 | 'b1' : 13, |
| 205 | 'b2' : 15, |
| 206 | 'b3' : 17, |
| 207 | 'b4' : 18, |
| 208 | 'c1' : 6, |
| 209 | 'c2' : 10, |
| 210 | 'c3' : 12, |
| 211 | 'storage' : 12 |
| 212 | } |
| 213 | ) |
|  | The cyclic delivery problem is the delivery problem described in Example |

#### 3.8 and shown in Figure 3.6 of the textbook. This is the same as acyclic delivery problem, but almost every arc also has its inverse.

searchProblem.py — (continued)

215 cyclic\_delivery\_problem = Search\_problem\_from\_explicit\_graph(

216 {'mail','ts','o103','o109','o111','b1','b2','b3','b4','c1','c2','c3',

217 'o125','o123','o119','r123','storage'},

218 [ Arc('ts','mail',6), Arc('mail','ts',6),

|  |  |
| --- | --- |
| 219 | Arc('o103','ts',8), Arc('ts','o103',8), |
| 220 | Arc('o103','b3',4), |
| 221 | Arc('o103','o109',12), Arc('o109','o103',12), |
| 222 | Arc('o109','o119',16), Arc('o119','o109',16), |
| 223 | Arc('o109','o111',4), Arc('o111','o109',4), |
| 224 | Arc('b1','c2',3), |
| 225 | Arc('b1','b2',6), Arc('b2','b1',6), |
| 226 | Arc('b2','b4',3), Arc('b4','b2',3), |
| 227 | Arc('b3','b1',4), Arc('b1','b3',4), |
| 228 | Arc('b3','b4',7), Arc('b4','b3',7), |
| 229 | Arc('b4','o109',7), |
| 230 | Arc('c1','c3',8), Arc('c3','c1',8), |
| 231 | Arc('c2','c3',6), Arc('c3','c2',6), |
| 232 | Arc('c2','c1',4), Arc('c1','c2',4), |
| 233 | Arc('o123','o125',4), Arc('o125','o123',4), |
| 234 | Arc('o123','r123',4), Arc('r123','o123',4), |
| 235 | Arc('o119','o123',9), Arc('o123','o119',9), |
| 236 | Arc('o119','storage',7), Arc('storage','o119',7)], |
| 237 | start = 'o103', |
| 238 | goals = {'r123'}, |
| 239 | hmap = { |
| 240 | 'mail' : 26, |
| 241 | 'ts' : 23, |
| 242 | 'o103' : 21, |
| 243 | 'o109' : 24, |
| 244 | 'o111' : 27, |
| 245 | 'o119' : 11, |
| 246 | 'o123' : 4, |
| 247 | 'o125' : 6, |
| 248 | 'r123' : 0, |
| 249 | 'b1' : 13, |
| 250 | 'b2' : 15, |
| 251 | 'b3' : 17, |
| 252 | 'b4' : 18, |
| 253 | 'c1' : 6, |
| 254 | 'c2' : 10, |
| 255 | 'c3' : 12, |
| 256 | 'storage' : 12 |
| 257 | } |
| 258 | ) |

## Generic Searcher and Variants

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

### Searcher

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

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

11 **from** display **import** Displayable, visualize

12

13 **class** Searcher(Displayable):

14 """returns a searcher for a problem.

15 Paths can be found by repeatedly calling search().

16 This does depth-first search unless overridden

17 """

18 **def** init (self, problem):

19 """creates a searcher from a problem

20 """

21 self.problem = problem

22 self.initialize\_frontier()

23 self.num\_expanded = 0

24 self.add\_to\_frontier(Path(problem.start\_node()))

25 **super**(). init ()

26

27 **def** initialize\_frontier(self):

28 self.frontier = []

29

30 **def** empty\_frontier(self):

31 **return** self.frontier == []

32

33 **def** add\_to\_frontier(self,path):

34 self.frontier.append(path)

35

36 @visualize

37 **def** search(self):

38 """returns (next) path from the problem's start node

39 to a goal node.

40 Returns None if no path exists.

41 """

42 **while not** self.empty\_frontier():

43 path = self.frontier.pop()

44 self.display(2, "Expanding:",path,"(cost:",path.cost,")")

45 self.num\_expanded += 1

46 **if** self.problem.is\_goal(path.end()): # solution found

47 self.display(1, self.num\_expanded, "paths have been expanded and",

48 **len**(self.frontier), "paths remain in the frontier")

49 self.solution = path # store the solution found

50 **return** path

51 **else**:

52 neighs = self.problem.neighbors(path.end())

53 self.display(3,"Neighbors are", neighs)

54 **for** arc **in reversed**(neighs):

55 self.add\_to\_frontier(Path(path,arc))

56 self.display(3,"Frontier:",self.frontier)

57 self.display(1,"No (more) solutions. Total of",

58 self.num\_expanded,"paths expanded.")

#### Note that this reverses the neigbours so that it implements depth-first search in an intutive manner (expanding the first neighbor first). This might not be required for other methods.

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

* + 1. Frontier as a Priority Queue

In many of the search algorithms, such as *A*∗ and other best-first searchers, the frontier is implemented as a priority queue. Here we use the Python’s built-in priority queue implementations, heapq.

Following the lead of the Python documentation, [http://docs.python.org/](http://docs.python.org/3.3/library/heapq.html) [3.3/library/heapq.html](http://docs.python.org/3.3/library/heapq.html), a frontier is a list of triples. The first element of each triple is the value to be minimized. The second element is a unique index which specifies the order when the first elements are the same, and the third element is the path that is on the queue. The use of the unique index ensures that the priority queue implementation does not compare paths; whether one path is less than another is not defined. It also lets us control what sort of search (e.g., depth-first or breadth-first) occurs when the value to be minimized does not give a unique next path.

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

searchGeneric.py — (continued)

60 **import** heapq # part of the Python standard library

61 **from** searchProblem **import** Path

62

63 **class** FrontierPQ(**object**):

64 """A frontier consists of a priority queue (heap), frontierpq, of

65 (value, index, path) triples, where

66 \* value is the value we want to minimize (e.g., path cost + h).

67 \* index is a unique index for each element

68 \* path is the path on the queue

69 Note that the priority queue always returns the smallest element.

70 """

71

72 **def** init (self):

73 """constructs the frontier, initially an empty priority queue

74 """

75 self.frontier\_index = 0 # the number of items ever added to the frontier

|  |  |  |
| --- | --- | --- |
| 76 |  | self.frontierpq = [] # the frontier priority queue |
| 77 |  |  |
| 78 | **def** | empty(self): |
| 79 |  | """is True if the priority queue is empty""" |
| 80 |  | **return** self.frontierpq == [] |
| 81 |  |  |
| 82 | **def** | add(self, path, value): |
| 83 |  | """add a path to the priority queue |
| 84 |  | value is the value to be minimized""" |
| 85 |  | self.frontier\_index += 1 # get a new unique index |
| 86 |  | heapq.heappush(self.frontierpq,(value, -self.frontier\_index, path)) |
| 87 |  |  |
| 88 | **def** | pop(self): |
| 89 |  | """returns and removes the path of the frontier with minimum value. |
| 90 |  | """ |
| 91 |  | (\_,\_,path) = heapq.heappop(self.frontierpq) |
| 92 |  | **return** path |

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

searchGeneric.py — (continued)

94 **def** count(self,val):

95 """returns the number of elements of the frontier with value=val"""

96 **return sum**(1 **for** e **in** self.frontierpq **if** e[0]==val)

97

98 **def** repr (self):

99 """string representation of the frontier"""

100 **return str**([(n,c,**str**(p)) **for** (n,c,p) **in** self.frontierpq])

101

102 **def** len (self):

103 """length of the frontier"""

104 **return len**(self.frontierpq)

105

106 **def** iter (self):

107 """iterate through the paths in the frontier"""

108 **for** (\_,\_,path) **in** self.frontierpq:

109 yield path

* + 1. *A*∗ Search

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

searchGeneric.py — (continued)

111 **class** AStarSearcher(Searcher):

112 """returns a searcher for a problem.

113 Paths can be found by repeatedly calling search().

114 """

115

116 **def** init (self, problem):

117 **super**(). init (problem)

118

119 **def** initialize\_frontier(self):

120 self.frontier = FrontierPQ()

121

122 **def** empty\_frontier(self):

123 **return** self.frontier.empty()

124

125 **def** add\_to\_frontier(self,path):

126 """add path to the frontier with the appropriate cost"""

127 value = path.cost+self.problem.heuristic(path.end())

128 self.frontier.add(path, value)

#### Testing:

searchGeneric.py — (continued)

130 **import** searchProblem as searchProblem

131

132 **def** test(SearchClass):

133 **print**("Testing problem 1:")

134 schr1 = SearchClass(searchProblem.problem1)

135 path1 = schr1.search()

136 **print**("Path found:",path1)

137 **assert list**(path1.nodes()) == ['g','d','c','b','a'], "Shortest path not found in problem1"

138 **print**("Passed unit test")

139

140 **if** name == " main ":

141 #test(Searcher)

142 test(AStarSearcher)

143

144 # example queries:

145 # searcher1 = Searcher(searchProblem.acyclic\_delivery\_problem) # DFS

146 # searcher1.search() # find first path

147 # searcher1.search() # find next path

148 # searcher2 = AStarSearcher(searchProblem.acyclic\_delivery\_problem) # A\*

149 # searcher2.search() # find first path

150 # searcher2.search() # find next path

151 # searcher3 = Searcher(searchProblem.cyclic\_delivery\_problem) # DFS

152 # searcher3.search() # find first path with DFS. What do you expect to happen?

153 # searcher4 = AStarSearcher(searchProblem.cyclic\_delivery\_problem) # A\*

154 # searcher4.search() # find first path

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

**Exercise 3.3** In the *add* method in *FrontierPQ* what does the ”-” in front of *frontier index* do? When there are multiple paths with the same *f* -value, which search method does this act like? What happens if the ”-” is removed? When there are multiple paths with the same value, which search method does this act like? Does it work better with or without the ”-”? What evidence did you base your conclusion on?

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

* + 1. Multiple Path Pruning

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

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

in *Searcher*.

searchMPP.py — Searcher with multiple-path pruning

11 **from** searchGeneric **import** AStarSearcher, visualize

12 **from** searchProblem **import** Path

13

14 **class** SearcherMPP(AStarSearcher):

15 """returns a searcher for a problem.

16 Paths can be found by repeatedly calling search().

17 """

18 **def** init (self, problem):

19 **super**(). init (problem)

20 self.explored = **set**()

21

22 @visualize

23 **def** search(self):

24 """returns next path from an element of problem's start nodes

25 to a goal node.

26 Returns None if no path exists.

27 """

28 **while not** self.empty\_frontier():

29 path = self.frontier.pop()

30 **if** path.end() **not in** self.explored:

31 self.display(2, "Expanding:",path,"(cost:",path.cost,")")

32 self.explored.add(path.end())

33 self.num\_expanded += 1

34 **if** self.problem.is\_goal(path.end()):

35 self.display(1, self.num\_expanded, "paths have been expanded and",

36 **len**(self.frontier), "paths remain in the frontier")

37 self.solution = path # store the solution found

38 **return** path

39 **else**:

40 neighs = self.problem.neighbors(path.end())

41 self.display(3,"Neighbors are", neighs)

42 **for** arc **in** neighs:

43 self.add\_to\_frontier(Path(path,arc))

44 self.display(3,"Frontier:",self.frontier)

45 self.display(1,"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** searchProblem

53 # searcherMPPcdp = SearcherMPP(searchProblem.cyclic\_delivery\_problem)

54 # print(searcherMPPcdp.search()) # find first path

**Exercise 3.5** Implement a searcher that implements cycle pruning instead of multiple-path pruning. You need to decide whether to check for cycles when paths are added to the frontier or when they are removed. (Hint: either method can be implemented by only changing one or two lines in SearcherMPP.) 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?

## Branch-and-bound Search

“searchBranchAndBound.py”, and copy and paste the example queries at the bottom of that file.

load

in folder “aipython”,

To run the demo,

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

searchBranchAndBound.py — Branch and Bound Search

11 **from** searchProblem **import** Path

12 **from** searchGeneric **import** Searcher

13 **from** display **import** Displayable, visualize

14

15 **class** DF\_branch\_and\_bound(Searcher):

16 """returns a branch and bound searcher for a problem.

17 An optimal path with cost less than bound can be found by calling search()

18 """

19 **def** init (self, problem, bound=**float**("inf")):

20 """creates a searcher than can be used with search() to find an optimal path.

21 bound gives the initial bound. By default this is infinite - meaning there

22 is no initial pruning due to depth bound

23 """

24 **super**(). init (problem)

25 self.best\_path = None

26 self.bound = bound

27

28 @visualize

29 **def** search(self):

30 """returns an optimal solution to a problem with cost less than bound.

31 returns None if there is no solution with cost less than bound."""

32 self.frontier = [Path(self.problem.start\_node())]

33 self.num\_expanded = 0

34 **while** self.frontier:

35 path = self.frontier.pop()

36 **if** path.cost+self.problem.heuristic(path.end()) < self.bound:

37 self.display(3,"Expanding:",path,"cost:",path.cost)

38 self.num\_expanded += 1

39 **if** self.problem.is\_goal(path.end()):

40 self.best\_path = path

41 self.bound = path.cost

42 self.display(2,"New best path:",path," cost:",path.cost)

43 **else**:

44 neighs = self.problem.neighbors(path.end())

45 self.display(3,"Neighbors are", neighs)

46 **for** arc **in reversed**(**list**(neighs)):

47 self.add\_to\_frontier(Path(path, arc))

48 self.display(1,"Number of paths expanded:",self.num\_expanded)

49 self.solution = self.best\_path

50 **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. Note that reversed only works on lists and tuples, but the neighbours can be generated.

Here is a unit test and some queries:

searchBranchAndBound.py — (continued)

52 **from** searchGeneric **import** test

53 **if** name == " main ":

54 test(DF\_branch\_and\_bound)

55

56 # Example queries:

57 **import** searchProblem

58 # searcherb1 = DF\_branch\_and\_bound(searchProblem.acyclic\_delivery\_problem)

59 # print(searcherb1.search()) # find optimal path

60 # searcherb2 = DF\_branch\_and\_bound(searchProblem.cyclic\_delivery\_problem, bound=100)

61 # print(searcherb2.search()) # find optimal path

**Exercise 3.6** Implement a branch-and-bound search uses recursion. Hint: you don’t need an explicit frontier, but can do a recursive call for the children.

**Exercise 3.7** After the branch-and-bound search found a solution, Sam ran search again, and noticed a different count. Sam hypothesized that this count was related to the number of nodes that an *A* search would use (either expand or be added to the frontier). Or maybe, Sam thought, the count for a number of nodes when the

∗

bound is slightly above the optimal path case is related to how *A*∗ would work. Is there relationship between these counts? Are there different things that it could count so they are related? Try to find the most specific statement that is true, and explain why it is true.

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

searchTest.py — code that may be useful to compare A\* and branch-and-bound

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 f-value=",asearcher.solution.cost)

26

27 **print**("\nA\* with MPP:"),

28 msearcher = SearcherMPP(problem)

29 **print**("Path found:",msearcher.search()," cost=",msearcher.solution.cost)

30 **print**("there are",msearcher.frontier.count(msearcher.solution.cost),

31 "elements remaining on the queue with f-value=",msearcher.solution.cost)

32

33 bound = asearcher.solution.cost+0.01

34 **print**("\nBranch and bound (with too-good initial bound of", bound,")")

35 tbb = DF\_branch\_and\_bound(problem,bound) # cheating!!!!

36 **print**("Path found:",tbb.search()," cost=",tbb.solution.cost)

37 **print**("Rerunning B&B")

38 **print**("Path found:",tbb.search())

39

40 bbound = asearcher.solution.cost\*2+10

41 **print**("\nBranch and bound (with not-very-good initial bound of", bbound, ")")

42 tbb2 = DF\_branch\_and\_bound(problem,bbound) # cheating!!!!

43 **print**("Path found:",tbb2.search()," cost=",tbb2.solution.cost)

44 **print**("Rerunning B&B")

45 **print**("Path found:",tbb2.search())

46

47 **print**("\nDepth-first search: (Use ˆC if it goes on forever)")

48 tsearcher = Searcher(problem)

49 **print**("Path found:",tsearcher.search()," cost=",tsearcher.solution.cost)

50

51

52 **import** searchProblem

53 **from** searchTest **import** run

54 **if** name == " main ":

55 run(searchProblem.problem1,"Problem 1")

56 # run(searchProblem.acyclic\_delivery\_problem,"Acyclic Delivery")

57 # run(searchProblem.cyclic\_delivery\_problem,"Cyclic Delivery")

58 # also test some graphs with cycles, and some with multiple least-cost paths

# Chapter 4

Reasoning with Constraints

## Constraint Satisfaction Problems

### Constraints

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

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

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

name property that gives a printable name of the function; built-in functions and functions that are defined using *def* have such a property; for other functions you may need to define this property.

cspProblem.py — Representations of a Constraint Satisfaction Problem

11 **class** Constraint(**object**):

12 """A Constraint consists of

13 \* scope: a tuple of variables

14 \* condition: a function that can applied to a tuple of values

15 for the variables

16 """

17 **def** init (self, scope, condition):

18 self.scope = scope

19 self.condition = condition

20

21 **def** repr (self):

22 **return** self.condition. name + **str**(self.scope)

#### 49

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

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

In Python, the notation is used for unpacking a tuple. For example,

∗

*F*( (1, 2, 3)) is the same as *F*(1, 2, 3). So if *t* has value (1, 2, 3), then *F*( *t*) is

∗ ∗

the same as *F*(1, 2, 3).

cspProblem.py — (continued)

24 **def** holds(self,assignment):

25 """returns the value of Constraint con evaluated in assignment.

26

27 precondition: all variables are assigned in assignment

28 """

29 **return** self.condition(\***tuple**(assignment[v] **for** v **in** self.scope))

### CSPs

#### A constraint satisfaction problem (CSP) requires:

* *domains*: a dictionary that maps variables to the set of possible values. Thus *domains*[*var*] is the domain of variable *var*.
* *constaraints*: a set or list of constraints. Other properties are inferred from these:
* *variables* is the set of variables. The variables can be enumerated by using “for var in domains” because iterating over a dictionary gives the keys, which in this case are the variables.
* *var to const* is a mapping from variables to set of constraints, such that

*var to const*[*var*] is the set of constraints with *var* in the scope.

cspProblem.py — (continued)

31 **class** CSP(**object**):

32 """A CSP consists of

33 \* domains, a dictionary that maps each variable to its domain

34 \* constraints, a list of constraints

35 \* variables, a set of variables

36 \* var\_to\_const, a variable to set of constraints dictionary

37 """

38 **def** init (self,domains,constraints):

39 """domains is a variable:domain dictionary

40 constraints is a list of constriants

|  |  |  |
| --- | --- | --- |
| 41  42  43  44  45  46  47  48 | """  self.variables = **set**(domains) self.domains = domains self.constraints = constraints  self.var\_to\_const = {var:**set**() **for** var **in** self.variables}  **for** con **in** constraints:  **for** var **in** con.scope: self.var\_to\_const[var].add(con) | |
| 49 |  |  |
| 50 | **def** | str (self): |
| 51 |  | """string representation of CSP""" |
| 52 |  | **return str**(self.domains) |
| 53 |  |  |
| 54 | **def** | repr (self): |
| 55 |  | """more detailed string representation of CSP""" |
| 56 |  | **return** "CSP("+**str**(self.domains)+", "+**str**([**str**(c) **for** c **in** self.constraints])+")" |

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

cspProblem.py — (continued)

58 **def** consistent(self,assignment):

59 """assignment is a variable:value dictionary

60 returns True if all of the constraints that can be evaluated

61 evaluate to True given assignment.

62 """

63 **return all**(con.holds(assignment)

64 **for** con **in** self.constraints

65 **if all**(v **in** assignment **for** v **in** con.scope))

### 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* = *y*. Allowing a function of multiple arguments to use its arguments one at a time is called **currying**, after the logician Haskell Curry. The use of a condition in constraints requires that the function with a single argument has a name.

ƒ

cspExamples.py — Example CSPs

11 **from** cspProblem **import** CSP, Constraint

12 **from** operator **import** lt,ne,eq,gt

13

14 **def** ne\_(val):

15 """not equal value"""

16 # nev = lambda x: x != val # alternative definition

17 # nev = partial(neq,val) # another alternative definition

18 **def** nev(x):

19 **return** val != x

20 nev. name = **str**(val)+"!=" # name of the function

21 **return** nev

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

cspExamples.py — (continued)

23 **def** is\_(val):

24 """is a value"""

25 # isv = lambda x: x == val # alternative definition

26 # isv = partial(eq,val) # another alternative definition

27 **def** isv(x):

28 **return** val == x

29 isv. name = **str**(val)+"=="

30 **return** isv

#### The CSP, *csp*0 has variables *X*, *Y* and *Z*, each with domain 1, 2, 3 . The con- straints are *X < Y* and *Y < Z*.

{ }

cspExamples.py — (continued)

32 csp0 = CSP({'X':{1,2,3},'Y':{1,2,3}, 'Z':{1,2,3}},

33 [ Constraint(('X','Y'),lt),

34 Constraint(('Y','Z'),lt)])

#### The CSP, *csp*1 has variables *A*, *B* and *C*, each with domain 1, 2, 3, 4 . The con- straints are *A < B*, *B* = 2 and *B < C*. This is slightly more interesting than *csp*0 as it has more solutions. This example is used in the unit tests, and so if it is changed, the unit tests need to be changed.

ƒ

{ }

cspExamples.py — (continued)

36 C0 = Constraint(('A','B'),lt)

37 C1 = Constraint(('B',),ne\_(2))

38 C2 = Constraint(('B','C'),lt)

39 csp1 = CSP({'A':{1,2,3,4},'B':{1,2,3,4}, 'C':{1,2,3,4}},

40 [C0, C1, C2])

#### The next CSP, *csp*2 is Example 4.9 of the textbook; the domain consistent network is shown in Figure [4.1.](#_bookmark55)

cspExamples.py — (continued)

42 csp2 = CSP({'A':{1,2,3,4},'B':{1,2,3,4}, 'C':{1,2,3,4},

43 'D':{1,2,3,4}, 'E':{1,2,3,4}},

44 [ Constraint(('B',),ne\_(3)),

45 Constraint(('C',),ne\_(2)),

46 Constraint(('A','B'),ne),

47 Constraint(('B','C'),ne),

48 Constraint(('C','D'),lt),

49 Constraint(('A','D'),eq),

50 Constraint(('A','E'),gt),

51 Constraint(('B','E'),gt),

52 Constraint(('C','E'),gt),

*A*

{1,2,3,4}

*B*

{1,2,4}

*D*

{1,2,3,4}

*C*

{1,3,4}

*E*

{1,2,3,4}

*C*  *D*

*A*  *B*

*E*  *D*

*E*  *C*

*E*  *A*

*E*  *B*

*B*  *D*

*B*  *C*

*A*  *D*

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

53 Constraint(('D','E'),gt),

54 Constraint(('B','D'),ne)])

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

cspExamples.py — (continued)

56 csp3 = CSP({'A':{1,2,3,4},'B':{1,2,3,4}, 'C':{1,2,3,4},

57 'D':{1,2,3,4}, 'E':{1,2,3,4}},

58 [Constraint(('A','B'), ne),

59 Constraint(('A','D'), lt),

60 Constraint(('A','E'), **lambda** a,e: (a-e)%2 == 1), # A-E is odd

61 Constraint(('B','E'), lt),

62 Constraint(('D','C'), lt),

63 Constraint(('C','E'), ne),

64 Constraint(('D','E'), ne)])

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

cspExamples.py — (continued)

66 **def** adjacent(x,y):

67 """True when x and y are adjacent numbers"""

**Words:**

4

3

2

1

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

Figure 4.2: A crossword puzzle to be solved

68 **return abs**(x-y) == 1

69

70 csp4 = CSP({'A':{1,2,3,4,5},'B':{1,2,3,4,5}, 'C':{1,2,3,4,5},

71 'D':{1,2,3,4,5}, 'E':{1,2,3,4,5}},

72 [Constraint(('A','B'),adjacent),

73 Constraint(('B','C'),adjacent),

74 Constraint(('C','D'),adjacent),

75 Constraint(('D','E'),adjacent),

76 Constraint(('A','C'),ne),

77 Constraint(('B','D'),ne),

78 Constraint(('C','E'),ne)])

#### The following examples represent the crossword shown in Figure [4.2.](#_bookmark56)

cspExamples.py — (continued)

80 **def** meet\_at(p1,p2):

81 """returns a function that is true when the words meet at the postions p1, p2

82 """

83 **def** meets(w1,w2):

84 **return** w1[p1] == w2[p2]

85 meets. name = "meet\_at("+**str**(p1)+','+**str**(p2)+')'

86 **return** meets

87

88 crossword1 = CSP({'one\_across':{'ant', 'big', 'bus', 'car', 'has'},

89 'one\_down':{'book', 'buys', 'hold', 'lane', 'year'},

90 'two\_down':{'ginger', 'search', 'symbol', 'syntax'},

91 'three\_across':{'book', 'buys', 'hold', 'land', 'year'},

92 'four\_across':{'ant', 'big', 'bus', 'car', 'has'}},

93 [Constraint(('one\_across','one\_down'),meet\_at(0,0)),

94 Constraint(('one\_across','two\_down'),meet\_at(2,0)),

95 Constraint(('three\_across','two\_down'),meet\_at(2,2)),

96 Constraint(('three\_across','one\_down'),meet\_at(0,2)),

97 Constraint(('four\_across','two\_down'),meet\_at(0,4))])

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

cspExamples.py — (continued)

99 words = {'ant', 'big', 'bus', 'car', 'has','book', 'buys', 'hold',

100 'lane', 'year', 'ginger', 'search', 'symbol', 'syntax'}

101

102 **def** is\_word(\*letters, words=words):

103 """is true if the letters concatenated form a word in words"""

104 **return** "".join(letters) **in** words

105

106 letters = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",

107 "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y",

108 "z"]

109

110 crossword1d = CSP({'p00':letters, 'p10':letters, 'p20':letters, # first row

111 'p01':letters, 'p21':letters, # second row

112 'p02':letters, 'p12':letters, 'p22':letters, 'p32':letters, # third row

113 'p03':letters, 'p23':letters, #fourth row

114 'p24':letters, 'p34':letters, 'p44':letters, # fifth row

115 'p25':letters # sixth row

116 },

117 [Constraint(('p00', 'p10', 'p20'), is\_word), #1-across

118 Constraint(('p00', 'p01', 'p02', 'p03'), is\_word), # 1-down

119 Constraint(('p02', 'p12', 'p22', 'p32'), is\_word), # 3-across

120 Constraint(('p20', 'p21', 'p22', 'p23', 'p24', 'p25'), is\_word), # 2-down

121 Constraint(('p24', 'p34', 'p44'), is\_word) # 4-across

122 ])

Unit tests

#### The following defines a unit test for solvers on example csp1.

cspExamples.py — (continued)

124 **def** test(CSP\_solver, csp=csp1,

125 solutions=[{'A': 1, 'B': 3, 'C': 4}, {'A': 2, 'B': 3, 'C': 4}]):

126 """CSP\_solver is a solver that finds a solution to a CSP.

127 CSP\_solver takes a csp and returns a solution.

128 csp has to be a CSP, where solutions is the list of all solutions.

129 This tests whether the solution returned by CSP\_solver is a solution.

130 """

131 **print**("Testing csp with",CSP\_solver. doc )

132 sol0 = CSP\_solver(csp)

133 **print**("Solution found:",sol0)

134 **assert** sol0 **in** solutions, "Solution not found for "+**str**(csp)

135 **print**("Passed unit test")

**Exercise 4.1** Modify *test* so that instead of taking in a list of solutions, it checks whether the returned solution actually is a solution.

**Exercise 4.2** Propose a test that is appropriate for CSPs with no solutions. As- sume that the test designer knows there are no solutions. Consider what a CSP solver should return if there are no solutions to the CSP.

## Solving a CSP using Search

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

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

* + - A node is a *variable* : *value* dictionary.

#### 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 de- pend on the context. If no variable ordering is given, this makes no at- tempt 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 **from** utilities **import** dict\_union

14

15 **class** Search\_from\_CSP(Search\_problem):

16 """A search problem directly from the CSP.

17

18 A node is a variable:value dictionary"""

19 **def** init (self, csp, variable\_order=None):

20 self.csp=csp

21 **if** variable\_order:

22 **assert set**(variable\_order) == **set**(csp.variables)

23 **assert len**(variable\_order) == **len**(csp.variables)

24 self.variables = variable\_order

25 **else**:

26 self.variables = **list**(csp.variables)

27

28 **def** is\_goal(self, node):

29 """returns whether the current node is a goal for the search

30 """

31 **return len**(node)==**len**(self.csp.variables)

32

33 **def** start\_node(self):

34 """returns the start node for the search

35 """

36 **return** {}

##### 4.2. Solving a CSP using Search 57

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.

cspSearch.py — (continued)

38 **def** neighbors(self, node):

39 """returns a list of the neighboring nodes of node.

40 """

41 var = self.variables[**len**(node)] # the next variable

42 res = []

43 **for** val **in** self.csp.domains[var]:

44 new\_env = dict\_union(node,{var:val}) #dictionary union

45 **if** self.csp.consistent(new\_env):

46 res.append(Arc(node,new\_env))

47 **return** res

cspSearch.py — (continued)

49 **from** cspExamples **import** csp1,csp2,test, crossword1, crossword1d

50 **from** searchGeneric **import** Searcher

51

52 **def** dfs\_solver(csp):

53 """depth-first search solver"""

54 path = Searcher(Search\_from\_CSP(csp)).search()

55 **if** path **is not** None:

56 **return** path.end()

57 **else**:

58 **return** None

59

60 **if** name == " main ":

61 test(dfs\_solver)

62

63 ## Test Solving CSPs with Search:

64 searcher1 = Searcher(Search\_from\_CSP(csp1))

65 #print(searcher1.search()) # get next solution

66 searcher2 = Searcher(Search\_from\_CSP(csp2))

67 #print(searcher2.search()) # get next solution

68 searcher3 = Searcher(Search\_from\_CSP(crossword1))

69 #print(searcher3.search()) # get next solution

70 searcher4 = Searcher(Search\_from\_CSP(crossword1d))

71 #print(searcher4.search()) # get next solution (warning: slow)

**Exercise 4.3** What would happen if we constructed the new assignment by as- signing *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 4.4** Change neighbors so that it returns an iterator of values rather than a list. (Hint: use *yield*.)

## Consistency Algorithms

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

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

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

11 **from** display **import** Displayable

12

13 **class** Con\_solver(Displayable):

14 """Solves a CSP with arc consistency and domain splitting

15 """

16 **def** init (self, csp, \*\*kwargs):

17 """a CSP solver that uses arc consistency

18 \* csp is the CSP to be solved

19 \* kwargs is the keyword arguments for Displayable superclass

20 """

21 self.csp = csp

22 **super**(). init (\*\*kwargs) # Or Displayable. init (self,\*\*kwargs)

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

cspConsistency.py — (continued)

24 **def** make\_arc\_consistent(self, orig\_domains=None, to\_do=None):

25 """Makes this CSP arc-consistent using generalized arc consistency

26 orig\_domains is the original domains

27 to\_do is a set of (variable,constraint) pairs

28 returns the reduced domains (an arc-consistent variable:domain dictionary)

29 """

30 **if** orig\_domains **is** None:

31 orig\_domains = self.csp.domains

32 **if** to\_do **is** None:

33 to\_do = {(var, const) **for** const **in** self.csp.constraints

34 **for** var **in** const.scope}

35 **else**:

36 to\_do = to\_do.copy() # use a copy of to\_do

37 domains = orig\_domains.copy()

38 self.display(2,"Performing AC with domains", domains)

39 **while** to\_do:

40 var, const = self.select\_arc(to\_do)

41 self.display(3, "Processing arc (", var, ",", const, ")")

42 other\_vars = [ov **for** ov **in** const.scope **if** ov != var]

43 **if len**(other\_vars)==0:

44 new\_domain = {val **for** val **in** domains[var]

45 **if** const.holds({var:val})}

46 **elif len**(other\_vars)==1:

47 other = other\_vars[0]

48 new\_domain = {val **for** val **in** domains[var]

49 **if any**(const.holds({var: val,other:other\_val})

50 **for** other\_val **in** domains[other])}

51 **else**: # general case

52 new\_domain = {val **for** val **in** domains[var]

53 **if** self.any\_holds(domains, const, {var: val}, other\_vars)}

54 **if** new\_domain != domains[var]:

55 self.display(4, "Arc: (", var, ",", const, ") is inconsistent")

56 self.display(3, "Domain pruned", "dom(", var, ") =", new\_domain,

57 " due to ", const)

58 domains[var] = new\_domain

59 add\_to\_do = self.new\_to\_do(var, const) - to\_do

60 to\_do |= add\_to\_do # set union

61 self.display(3, " adding", add\_to\_do **if** add\_to\_do **else** "nothing", "to to\_do.")

62 self.display(4, "Arc: (", var, ",", const, ") now consistent")

63 self.display(2, "AC done. Reduced domains", domains)

64 **return** domains

65

66 **def** new\_to\_do(self, var, const):

67 """returns new elements to be added to to\_do after assigning

68 variable var in constraint const.

69 """

70 **return** {(nvar, nconst) **for** nconst **in** self.csp.var\_to\_const[var]

71 **if** nconst != const

72 **for** nvar **in** nconst.scope

73 **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 se- lects which ever element *pop* method for sets returns. A user interface could allow the user to select an arc. Alternatively a more sophisticated selection could be employed (or just a stack or a queue).

cspConsistency.py — (continued)

75 **def** select\_arc(self, to\_do):

76 """Selects the arc to be taken from to\_do .

77 \* to\_do is a set of arcs, where an arc is a (variable,constraint) pair

78 the element selected must be removed from to\_do.

79 """

80 **return** to\_do.pop()

#### The function *any holds* is useful to go beyond unary and binary constraints. It allows us to use constraints involving an arbitrary number of variables. (Note that it also works for unary and binary constraints; the cases where *len*(*other vars*) is 0 or 1 are not actually required, but are there for efficiency and because they are easier to understand.) *any holds* is a recursive function that tries to finds an assignment of values to the other variables (*other vars*) that satisfies constraint *const* given the assignment in *env*. The integer variable *ind* specifies which in-

dex to *other vars* needs to be checked next. As soon as one assignment returns *True*, the algorithm returns *True*. Note that it has side effects with respect to *env*; it changes the values of the variables in *other vars*. It should only be called when the side effects have no ill effects.

cspConsistency.py — (continued)

82 **def** any\_holds(self, domains, const, env, other\_vars, ind=0):

83 """returns True if Constraint const holds for an assignment

84 that extends env with the variables in other\_vars[ind:]

85 env is a dictionary

86 Warning: this has side effects and changes the elements of env

87 """

88 **if** ind == **len**(other\_vars):

89 **return** const.holds(env)

90 **else**:

91 var = other\_vars[ind]

92 **for** val **in** domains[var]:

93 # env = dict\_union(env,{var:val}) # no side effects!

94 env[var] = val

95 **if** self.any\_holds(domains, const, env, other\_vars, ind + 1):

96 **return** True

97 **return** False

### Direct Implementation of Domain Splitting

#### The following is a direct implementation of domain splitting with arc consis- tency that uses recursion. It finds one solution if one exists or returns False if there are no solutions.

cspConsistency.py — (continued)

99 **def** solve\_one(self, domains=None, to\_do=None):

100 """return a solution to the current CSP or False if there are no solutions

101 to\_do is the list of arcs to check

102 """

103 **if** domains **is** None:

104 domains = self.csp.domains

105 new\_domains = self.make\_arc\_consistent(domains, to\_do)

106 **if any**(**len**(new\_domains[var]) == 0 **for** var **in** domains):

107 **return** False

108 **elif all**(**len**(new\_domains[var]) == 1 **for** var **in** domains):

109 self.display(2, "solution:", {var: select(

110 new\_domains[var]) **for** var **in** new\_domains})

111 **return** {var: select(new\_domains[var]) **for** var **in** domains}

112 **else**:

113 var = self.select\_var(x **for** x **in** self.csp.variables **if len**(new\_domains[x]) > 1)

114 **if** var:

115 dom1, dom2 = partition\_domain(new\_domains[var])

116 self.display(3, "...splitting", var, "into", dom1, "and", dom2)

117 new\_doms1 = copy\_with\_assign(new\_domains, var, dom1)

118 new\_doms2 = copy\_with\_assign(new\_domains, var, dom2)

119 to\_do = self.new\_to\_do(var, None)

120 self.display(3, " adding", to\_do **if** to\_do **else** "nothing", "to to\_do.")

121 **return** self.solve\_one(new\_doms1, to\_do) **or** self.solve\_one(new\_doms2, to\_do)

122

123 **def** select\_var(self, iter\_vars):

124 """return the next variable to split"""

125 **return** select(iter\_vars)

126

127 **def** partition\_domain(dom):

128 """partitions domain dom into two.

129 """

130 split = **len**(dom) // 2

131 dom1 = **set**(**list**(dom)[:split])

132 dom2 = dom - dom1

133 **return** dom1, dom2

#### The domains are implemented as a dictionary that maps each variables to its domain. Assigning a value in Python has side effects which we want to avoid. *copy with assign* takes a copy of the domains dictionary, perhaps al- lowing for a new domain for a variable. It creates a copy of the CSP with an (optional) assignment of a new domain to a variable. Only the domains are copied.

cspConsistency.py — (continued)

135 **def** copy\_with\_assign(domains, var=None, new\_domain={True, False}):

136 """create a copy of the domains with an assignment var=new\_domain

137 if var==None then it is just a copy.

138 """

139 newdoms = domains.copy()

140 **if** var **is not** None:

141 newdoms[var] = new\_domain

142 **return** newdoms

cspConsistency.py — (continued)

144 **def** select(iterable):

145 """select an element of iterable. Returns None if there is no such element.

146

147 This implementation just picks the first element.

148 For many of the uses, which element is selected does not affect correctness,

149 but may affect efficiency.

150 """

151 **for** e **in** iterable:

152 **return** e # returns first element found

**Exercise 4.5** Implement of *solve all* that is like *solve one* but returns the set of all solutions.

**Exercise 4.6** Implement *solve enum* that enumerates the solutions. It should use Python’s *yield* (and perhaps *yield from*).

#### Unit test:

cspConsistency.py — (continued)

154 **from** cspExamples **import** test

155 **def** ac\_solver(csp):

156 "arc consistency (solve\_one)"

157 **return** Con\_solver(csp).solve\_one()

158 **if** name == " main ":

159 test(ac\_solver)

### 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.](#_bookmark33)

A node is domains dictionary.

cspConsistency.py — (continued)

161 **from** searchProblem **import** Arc, Search\_problem

162

163 **class** Search\_with\_AC\_from\_CSP(Search\_problem,Displayable):

|  |  |  |
| --- | --- | --- |
| 164 | """A search problem with arc consistency and domain splitting | |
| 165 |  | |
| 166 | A node is a CSP """ | |
| 167 | **def** init (self, csp): | |
| 168 | self.cons = Con\_solver(csp) #copy of the CSP | |
| 169  170 | self.domains = self.cons.make\_arc\_consistent() | |
| 171 | **def** | is\_goal(self, node): |
| 172 |  | """node is a goal if all domains have 1 element""" |
| 173 |  | **return all**(**len**(node[var])==1 **for** var **in** node) |
| 174 |  |  |
| 175 | **def** | start\_node(self): |
| 176 |  | **return** self.domains |
| 177 |  |  |
| 178 | **def** | neighbors(self,node): |
| 179 |  | """returns the neighboring nodes of node. |
| 180 |  | """ |
| 181 |  | neighs = [] |
| 182 |  | var = select(x **for** x **in** node **if len**(node[x])>1) |
| 183 |  | **if** var: |

184 dom1, dom2 = partition\_domain(node[var])

185 self.display(2,"Splitting", var, "into", dom1, "and", dom2)

186 to\_do = self.cons.new\_to\_do(var,None)

187 **for** dom **in** [dom1,dom2]:

188 newdoms = copy\_with\_assign(node,var,dom)

189 cons\_doms = self.cons.make\_arc\_consistent(newdoms,to\_do)

190 **if all**(**len**(cons\_doms[v])>0 **for** v **in** cons\_doms):

191 # all domains are non-empty

192 neighs.append(Arc(node,cons\_doms))

193 **else**:

194 self.display(2,"...",var,"in",dom,"has no solution")

195 **return** neighs

**Exercise 4.7** When splitting a domain, this code splits the domain into half, ap- proximately 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)

197 **from** cspExamples **import** test

198 **from** searchGeneric **import** Searcher

199

200 **def** ac\_search\_solver(csp):

201 """arc consistency (search interface)"""

202 sol = Searcher(Search\_with\_AC\_from\_CSP(csp)).search()

203 **if** sol:

204 **return** {v:select(d) **for** (v,d) **in** sol.end().items()}

205

206 **if** name == " main ":

207 test(ac\_search\_solver)

#### Testing:

|  |  |  |
| --- | --- | --- |
|  |  | cspConsistency.py — (continued) |
| 209 |  | **from** cspExamples **import** csp1, csp2, crossword1, crossword1d |
| 210 |  |  |
| 211 |  | ## Test Solving CSPs with Arc consistency and domain splitting: |
| 212 |  | #Con\_solver.max\_display\_level = 4 # display details of AC (0 turns off) |
| 213 |  | #Con\_solver(csp1).solve\_one() |
| 214 |  | #searcher1d = Searcher(Search\_with\_AC\_from\_CSP(csp1)) |
| 215 |  | #print(searcher1d.search()) |
| 216 |  | #Searcher.max\_display\_level = 2 # display search trace (0 turns off) |
| 217 |  | #searcher2c = Searcher(Search\_with\_AC\_from\_CSP(csp2)) |
| 218 |  | #print(searcher2c.search()) |
| 219 |  | #searcher3c = Searcher(Search\_with\_AC\_from\_CSP(crossword1)) |
| 220 |  | #print(searcher3c.search()) |
| 221 |  | #searcher5c = Searcher(Search\_with\_AC\_from\_CSP(crossword1d)) |
| 222 |  | #print(searcher5c.search()) |

## 4.4 Solving CSPs using Stochastic Local Search

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

This implements both the two-stage choice, the any-conflict algorithm and a random choice of variable (and a probabilistic mix of the three).

Given a CSP, the stochastic local searcher (*SLSearcher*) creates the data struc- tures:

* + - * *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 in- volved 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**(self.csp.domains[var]) > 1}

25 # Create assignment and conflicts set

26 self.current\_assignment = None # this will trigger a random restart

27 self.number\_of\_steps = 1 #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\_sample(dom) **for**

33 (var,dom) **in** self.csp.domains.items()}

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.

This method selects one of two implementations. The argument *pob best* is the probability of selecting a best variable (one involving the most conflicts). When the value of *prob best* is positive, the algorithm needs to maintain a prior- ity queue of variables and the number of conflicts (using *search with var pq*). If the probability of selecting a best variable is zero, it does not need to maintain this priority queue (as implemented in *search with any conflict*).

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

− −

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

cspSLS.py — (continued)

42 **def** search(self,max\_steps, prob\_best=0, prob\_anycon=1.0):

43 """

44 returns the number of steps or None if these 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 up

48 prob\_best is the probability that a best varaible (one in most conflict) is selected

49 prob\_anycon is the probability that a variabe 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 **return** self.number\_of\_steps

57 **if** prob\_best > 0: # we need to maintain a variable priority queue

58 **return** self.search\_with\_var\_pq(max\_steps, prob\_best, prob\_anycon)

59 **else**:

60 **return** self.search\_with\_any\_conflict(max\_steps, prob\_anycon)

**Exercise 4.8** 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*).

* + 1. Any-conflict

If the probability of picking a best variable is zero, the implementation need to keeps track of which variables are in conflicts.

cspSLS.py — (continued)

62 **def** search\_with\_any\_conflict(self, max\_steps, prob\_anycon=1.0):

63 """Searches with the any\_conflict heuristic.

64 This relies on just maintaining the set of conflicts;

65 it does not maintain a priority queue

66 """

67 self.variable\_pq = None # we are not maintaining the priority queue.

68 # This ensures it is regenerated if needed.

69 **for** i **in range**(max\_steps):

70 self.number\_of\_steps +=1

71 **if** random.random() < prob\_anycon:

72 con = random\_sample(self.conflicts) # pick random conflict

73 var = random\_sample(con.scope) # pick variable in conflict

74 **else**:

75 var = random\_sample(self.variables\_to\_select)

76 **if len**(self.csp.domains[var]) > 1:

77 val = random\_sample(self.csp.domains[var] -

78 {self.current\_assignment[var]})

79 self.display(2,self.number\_of\_steps,": Assigning",var,"=",val)

80 self.current\_assignment[var]=val

81 **for** varcon **in** self.csp.var\_to\_const[var]:

82 **if** varcon.holds(self.current\_assignment):

83 **if** varcon **in** self.conflicts:

84 self.conflicts.remove(varcon)

85 **else**:

86 **if** varcon **not in** self.conflicts:

87 self.conflicts.add(varcon)

88 self.display(2," Number of conflicts",**len**(self.conflicts))

89 **if not** self.conflicts:

90 self.display(1,"Solution found:", self.current\_assignment,

91 "in", self.number\_of\_steps,"steps")

92 **return** self.number\_of\_steps

93 self.display(1,"No solution in",self.number\_of\_steps,"steps",

94 **len**(self.conflicts),"conflicts remain")

95 **return** None

**Exercise 4.9** This makes no attempt to find the best alternative value for a vari- able. Modify the code so that after selecting a variable it selects a value the reduces the number of conflicts by the most. Have a parameter that specifies the probabil- ity that the best value is chosen.

* + 1. Two-Stage Choice

This is the top-level searching algorithm that maintains a priority queue of vari- ables ordered by (the negative of) the number of conflicts, so that the variable

with the most conflicts is selected first. If there is no current priority queue of variables, one is created.

The main complexity here is to maintain the priority queue. This uses the dictionary *var differential* which specifies how much the values of variables should change. This is used with the updatable queue (page [68)](#_bookmark70) to find a vari- able with the most conflicts.

cspSLS.py — (continued)

97 **def** search\_with\_var\_pq(self,max\_steps, prob\_best=1.0, prob\_anycon=1.0):

98 """search with a priority queue of variables.

99 This is used to select a variable with the most conflicts.

100 """

101 **if not** self.variable\_pq:

102 self.create\_pq()

103 pick\_best\_or\_con = prob\_best + prob\_anycon

104 **for** i **in range**(max\_steps):

105 self.number\_of\_steps +=1

106 randnum = random.random()

107 ## Pick a variable

108 **if** randnum < prob\_best: # pick best variable

109 var,oldval = self.variable\_pq.top()

110 **elif** randnum < pick\_best\_or\_con: # pick a variable in a conflict

111 con = random\_sample(self.conflicts)

112 var = random\_sample(con.scope)

113 **else**: #pick any variable that can be selected

114 var = random\_sample(self.variables\_to\_select)

115 **if len**(self.csp.domains[var]) > 1: # var has other values

116 ## Pick a value

117 val = random\_sample(self.csp.domains[var] -

118 {self.current\_assignment[var]})

119 self.display(2,"Assigning",var,val)

120 ## Update the priority queue

121 var\_differential = {}

122 self.current\_assignment[var]=val

123 **for** varcon **in** self.csp.var\_to\_const[var]:

124 self.display(3,"Checking",varcon)

125 **if** varcon.holds(self.current\_assignment):

126 **if** varcon **in** self.conflicts: #was incons, now consis

127 self.display(3,"Became consistent",varcon)

128 self.conflicts.remove(varcon)

129 **for** v **in** varcon.scope: # v is in one fewer conflicts

130 var\_differential[v] = var\_differential.get(v,0)-1

131 **else**:

132 **if** varcon **not in** self.conflicts: # was consis, not now

133 self.display(3,"Became inconsistent",varcon)

134 self.conflicts.add(varcon)

135 **for** v **in** varcon.scope: # v is in one more conflicts

136 var\_differential[v] = var\_differential.get(v,0)+1

137 self.variable\_pq.update\_each\_priority(var\_differential)

138 self.display(2,"Number of conflicts",**len**(self.conflicts))

139 **if not** self.conflicts: # no conflicts, so solution found

140 self.display(1,"Solution found:", self.current\_assignment,"in",

141 self.number\_of\_steps,"steps")

142 **return** self.number\_of\_steps

143 self.display(1,"No solution in",self.number\_of\_steps,"steps",

144 **len**(self.conflicts),"conflicts remain")

145 **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 vari- ables 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)

147 **def** create\_pq(self):

148 """Create the variable to number-of-conflicts priority queue.

149 This is needed to select the variable in the most conflicts.

150

151 The value of a variable in the priority queue is the negative of the

152 number of conflicts the variable appears in.

153 """

154 self.variable\_pq = Updatable\_priority\_queue()

155 var\_to\_number\_conflicts = {}

156 **for** con **in** self.conflicts:

157 **for** var **in** con.scope:

158 var\_to\_number\_conflicts[var] = var\_to\_number\_conflicts.get(var,0)+1

159 **for** var,num **in** var\_to\_number\_conflicts.items():

160 **if** num>0:

161 self.variable\_pq.add(var,-num)

cspSLS.py — (continued)

163 **def** random\_sample(st):

164 """selects a random element from set st"""

165 **return** random.sample(st,1)[0]

**Exercise 4.10** This makes no attempt to find the best alternative value for a vari- able. Modify the code so that after selecting a variable it selects a value the reduces the number of conflicts by the most. Have a parameter that specifies the probabil- ity that the best value is chosen.

**Exercise 4.11** These implementations always select a value for the variable se- lected 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)?

* + 1. Updatable Priority Queues

An **updatable priority queue** is a priority queue, where key-value pairs can be stored, and the pair with the smallest key can be found and removed quickly,

and where the values can be updated. This implementation follows the idea of <http://docs.python.org/3.5/library/heapq.html>, where the updated ele- ments 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 sec- ond element is a random number. Note that Python requires this to be a list, not a tuple, as the tuple cannot be modified.

|  |  |  |  |
| --- | --- | --- | --- |
| 167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182 |  | cspSLS.py — (continued)  **class** Updatable\_priority\_queue(**object**):  """A priority queue where the values can be updated. Elements with the same value are ordered randomly.  This code is based on the ideas described in <http://docs.python.org/3.3/library/heapq.html> It could probably be done more efficiently by shuffling the modified element in the heap. """  **def** init (self):  self.pq = [] # priority queue of [val,rand,elt] triples self.elt\_map = {} # map from elt to [val,rand,elt] triple in pq self.REMOVED = "\*removed\*" # a string that won't be a legal element self.max\_size=0  **def** add(self,elt,val): | |
| 183 |  |  | """adds elt to the priority queue with priority=val. |
| 184 |  |  | """ |
| 185 |  |  | **assert** val <= 0,val |
| 186 |  |  | **assert** elt **not in** self.elt\_map, elt |
| 187 |  |  | new\_triple = [val, random.random(),elt] |
| 188 |  |  | heapq.heappush(self.pq, new\_triple) |
| 189 |  |  | self.elt\_map[elt] = new\_triple |
| 190 |  |  |  |
| 191 |  | **def** | remove(self,elt): |
| 192 |  |  | """remove the element from the priority queue""" |
| 193 |  |  | **if** elt **in** self.elt\_map: |
| 194 |  |  | self.elt\_map[elt][2] = self.REMOVED |
| 195 |  |  | **del** self.elt\_map[elt] |
| 196 |  |  |  |
| 197 |  | **def** | update\_each\_priority(self,update\_dict): |
| 198 |  |  | """update values in the priority queue by subtracting the values in |
| 199 |  |  | update\_dict from the priority of those elements in priority queue. |
| 200 |  |  | """ |
| 201 |  |  | **for** elt,incr **in** update\_dict.items(): |
| 202  203 |  |  | **if** incr != 0:  newval = self.elt\_map.get(elt,[0])[0] - incr |

|  |  |  |
| --- | --- | --- |
| 204 | **assert** newval <= 0, **str**(elt)+":"+**str**(newval+incr)+"-"+**str**(incr) | |
| 205 | self.remove(elt) | |
| 206 | **if** newval != 0: | |
| 207 | self.add(elt,newval) | |
| 208 |  | |
| 209  210  211  212  213  214  215  216  217  218  219 | **def** pop(self):  """Removes and returns the (elt,value) pair with minimal value. If the priority queue is empty, IndexError is raised.  """  self.max\_size = **max**(self.max\_size, **len**(self.pq)) # keep statistics triple = heapq.heappop(self.pq)  **while** triple[2] == self.REMOVED: triple = heapq.heappop(self.pq)  **del** self.elt\_map[triple[2]]  **return** triple[2], triple[0] # elt, value | |
| 220 | **def** | top(self): |
| 221 |  | """Returns the (elt,value) pair with minimal value, without removing it. |
| 222 |  | If the priority queue is empty, IndexError is raised. |
| 223 |  | """ |
| 224 |  | self.max\_size = **max**(self.max\_size, **len**(self.pq)) # keep statistics |
| 225 |  | triple = self.pq[0] |
| 226 |  | **while** triple[2] == self.REMOVED: |
| 227 |  | heapq.heappop(self.pq) |
| 228 |  | triple = self.pq[0] |
| 229 |  | **return** triple[2], triple[0] # elt, value |
| 230 |  |  |
| 231 | **def** | empty(self): |
| 232 |  | """returns True iff the priority queue is empty""" |
| 233 |  | **return all**(triple[2] == self.REMOVED **for** triple **in** self.pq) |

* + 1. Plotting Runtime Distributions

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

cspSLS.py — (continued)

235 **import** matplotlib.pyplot as plt

236

237 **class** Runtime\_distribution(**object**):

238 **def** init (self, csp, xscale='log'):

|  |  |
| --- | --- |
| 239 | """Sets up plotting for csp |
| 240 | xscale is either 'linear' or 'log' |
| 241 | """ |
| 242 | self.csp = csp |

|  |  |  |
| --- | --- | --- |
| 243 |  | plt.ion() |
| 244 |  | plt.xlabel("Number of Steps") |
| 245 |  | plt.ylabel("Cumulative Number of Runs") |
| 246 |  | plt.xscale(xscale) # Makes a 'log' or 'linear' scale |
| 247 |  |  |
| 248 | **def** | plot\_runs(self,num\_runs=100,max\_steps=1000, prob\_best=1.0, prob\_anycon=1.0): |
| 249 |  | """Plots num\_runs of SLS for the given settings. |
| 250 |  | """ |
| 251 |  | stats = [] |
| 252 |  | SLSearcher.max\_display\_level, temp\_mdl = 0, SLSearcher.max\_display\_level # no display |
| 253 |  | **for** i **in range**(num\_runs): |
| 254 |  | searcher = SLSearcher(self.csp) |
| 255 |  | num\_steps = searcher.search(max\_steps, prob\_best, prob\_anycon) |
| 256 |  | **if** num\_steps: |
| 257 |  | stats.append(num\_steps) |
| 258 |  | stats.sort() |
| 259 |  | **if** prob\_best >= 1.0: |
| 260 |  | label = "P(best)=1.0" |
| 261 |  | **else**: |
| 262 |  | p\_ac = **min**(prob\_anycon, 1-prob\_best) |
| 263 |  | label = "P(best)=%.2f, P(ac)=%.2f" % (prob\_best, p\_ac) |
| 264 |  | plt.plot(stats,**range**(**len**(stats)),label=label) |
| 265 |  | plt.legend(loc="upper left") |
| 266 |  | #plt.draw() |
| 267 |  | SLSearcher.max\_display\_level= temp\_mdl #restore display |

### Testing

cspSLS.py — (continued)

269 **from** cspExamples **import** test

270 **def** sls\_solver(csp,prob\_best=0.7):

271 """stochastic local searcher (prob\_best=0.7)"""

272 se0 = SLSearcher(csp)

273 se0.search(1000,prob\_best)

274 **return** se0.current\_assignment

275 **def** any\_conflict\_solver(csp):

276 """stochastic local searcher (any-conflict)"""

277 **return** sls\_solver(csp,0)

278

279 **if** name == " main ":

280 test(sls\_solver)

281 test(any\_conflict\_solver)

282

283 **from** cspExamples **import** csp1, csp2, crossword1

284

285 ## Test Solving CSPs with Search:

286 #se1 = SLSearcher(csp1); print(se1.search(100))

287 #se2 = SLSearcher(csp2); print(se2search(1000,1.0)) # greedy

288 #se2 = SLSearcher(csp2); print(se2.search(1000,0)) # any\_conflict

289 #se2 = SLSearcher(csp2); print(se2.search(1000,0.7)) # 70% greedy; 30% any\_conflict

290 #SLSearcher.max\_display\_level=2 #more detailed display

291 #se3 = SLSearcher(crossword1); print(se3.search(100),0.7)

292 #p = Runtime\_distribution(csp2)

293 #p.plot\_runs(1000,1000,0) # any\_conflict

294 #p.plot\_runs(1000,1000,1.0) # greedy

295 #p.plot\_runs(1000,1000,0.7) # 70% greedy; 30% any\_conflict

**Exercise 4.12** Modify this to plot the runtime, instead of the number of steps. To measure runtime use *timeit* ([https://docs.python.org/3.5/library/timeit.](https://docs.python.org/3.5/library/timeit.html) [html](https://docs.python.org/3.5/library/timeit.html)). Small runtimes are inaccurate, so timeit can run the same code multi- ple times. Stochastic local algorithms give different runtimes each time called. To make the timing meaningful, you need to make sure the random seed is the same for each repeated call (see random.getstate and random.setstate in [https:](https://docs.python.org/3.5/library/random.html)

[//docs.python.org/3.5/library/random.html](https://docs.python.org/3.5/library/random.html)). Because the runtime for different seeds can vary a great deal, for each seed, you should start with 1 iteration and multiplying it by, say 10, until the time is greater than 0.2 seconds. Make sure you plot the average time for each run. Before you start, try to estimate the total run- time, so you will be able to tell if there is a problem with the algorithm stopping.

Chapter 5

Propositions and Inference

## 5.1 Representing Knowledge Bases

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

logicProblem.py — Representations Logics

11 **class** Clause(**object**):

12 """A definite clause"""

13

14 **def** init (self,head,body=[]):

15 """clause with atom head and lost of atoms body"""

16 self.head=head

17 self.body = body

18

19 **def** str (self):

20 """returns the string representation of a clause.

21 """

22 **if** self.body:

23 **return** self.head + " <- " + " & ".join(self.body) + "."

24 **else**:

25 **return** self.head + "."

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

logicProblem.py — (continued)

27 **class** Askable(**object**):

28 """An askable atom"""

29

30 **def** init (self,atom):

31 """clause with atom head and lost of atoms body"""

#### 73

32 self.atom=atom

33

34 **def** str (self):

35 """returns the string representation of a clause."""

36 **return** "askable " + self.atom + "."

37

38 **def** yes(ans):

39 """returns true if the answer is yes in some form"""

40 **return** ans.lower() **in** ['yes', 'yes.', 'oui', 'oui.', 'y', 'y.'] # bilingual

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

logicProblem.py — (continued)

42 **from** display **import** Displayable

43

44 **class** KB(Displayable):

45 """A knowledge base consists of a set of clauses.

46 This also creates a dictionary to give fast access to the clauses with an atom in head.

47 """

48 **def** init (self, statements=[]):

49 self.statements = statements

50 self.clauses = [c **for** c **in** statements **if isinstance**(c, Clause)]

51 self.askables = [c.atom **for** c **in** statements **if isinstance**(c, Askable)]

52 self.atom\_to\_clauses = {} # dictionary giving clauses with atom as head

53 **for** c **in** self.clauses:

54 **if** c.head **in** self.atom\_to\_clauses:

55 self.atom\_to\_clauses[c.head].add(c)

56 **else**:

57 self.atom\_to\_clauses[c.head] = {c}

58

59 **def** clauses\_for\_atom(self,a):

60 """returns set of clauses with atom a as the head"""

61 **if** a **in** self.atom\_to\_clauses:

62 **return** self.atom\_to\_clauses[a]

63 **else**:

64 **return set**()

|  |  |  |
| --- | --- | --- |
| 65 |  | |
| 66 | **def** | str (self): |
| 67 |  | """returns a string representation of this knowledge base. |
| 68 |  | """ |
| 69 |  | **return** '\n'.join([**str**(c) **for** c **in** self.statements]) |

#### Here is a trivial example (I think therefore I am) using in the unit tests:

logicProblem.py — (continued)

71 triv\_KB = KB([

72 Clause('i\_am', ['i\_think']),

73 Clause('i\_think'),

74 Clause('i\_smell', ['i\_exist'])

75 ])

##### Bottom-up Proofs 75

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

logicProblem.py — (continued)

77 elect = KB([

78 Clause('light\_l1'),

79 Clause('light\_l2'),

80 Clause('ok\_l1'),

81 Clause('ok\_l2'),

82 Clause('ok\_cb1'),

83 Clause('ok\_cb2'),

84 Clause('live\_outside'),

85 Clause('live\_l1', ['live\_w0']),

86 Clause('live\_w0', ['up\_s2','live\_w1']),

87 Clause('live\_w0', ['down\_s2','live\_w2']),

88 Clause('live\_w1', ['up\_s1', 'live\_w3']),

89 Clause('live\_w2', ['down\_s1','live\_w3' ]),

90 Clause('live\_l2', ['live\_w4']),

91 Clause('live\_w4', ['up\_s3','live\_w3' ]),

92 Clause('live\_p\_1', ['live\_w3']),

93 Clause('live\_w3', ['live\_w5', 'ok\_cb1']),

94 Clause('live\_p\_2', ['live\_w6']),

95 Clause('live\_w6', ['live\_w5', 'ok\_cb2']),

96 Clause('live\_w5', ['live\_outside']),

97 Clause('lit\_l1', ['light\_l1', 'live\_l1', 'ok\_l1']),

98 Clause('lit\_l2', ['light\_l2', 'live\_l2', 'ok\_l2']),

99 Askable('up\_s1'),

100 Askable('down\_s1'),

101 Askable('up\_s2'),

102 Askable('down\_s2'),

103 Askable('up\_s3'),

104 Askable('down\_s2')

105 ])

106

107 # print(kb)

## Bottom-up Proofs

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

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

11 **from** logicProblem **import** yes

12

13 **def** fixed\_point(kb):

14 """Returns the fixed point of knowledge base kb.

15 """

16 fp = ask\_askables(kb)

17 added = True

18 **while** added:

19 added = False # added is true when an atom was added to fp this iteration

20 **for** c **in** kb.clauses:

21 **if** c.head **not in** fp **and all**(b **in** fp **for** b **in** c.body):

22 fp.add(c.head)

23 added = True

24 kb.display(2,c.head,"added to fp due to clause",c)

25 **return** fp

26

27 **def** ask\_askables(kb):

28 **return** {at **for** at **in** kb.askables **if** yes(**input**("Is "+at+" true? "))}

#### Testing:

logicBottomUp.py — (continued)

30 **from** logicProblem **import** triv\_KB

31 **def** test():

32 fp = fixed\_point(triv\_KB)

33 **assert** fp == {'i\_am','i\_think'}, "triv\_KB gave result "+**str**(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. Imple- ment ask-the-user so that questions are only asked if useful, and are not re-asked. For example, if there is a clause *h a b c d e*, where *c* and *e* are askable, *c* and *e* only need to be asked if *a*, *b*, *d* are all in *fp* and they have not been asked before. Askable *e* only needs to be asked if the user says “yes” to *c*. Askable *c* doesn’t need to be asked if the user previously replied “no” to *e*.

← ∧ ∧ ∧ ∧

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

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

**Exercise 5.3** It is possible to be asymptitocally more efficient (in terms of *b*) 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 b c d*, needs only be considered when *b* is added to *fp*. Once *b* is added to *fp*, if *c* is already in *pf* , we know that *a* can be added as soon as *d* is added. Implement this. What is its complexity as a function of *n* and *b*, the maximum number of atoms in the body of a clause?

← ∧ ∧

* 1. *Top-down Proofs* 77

## Top-down Proofs

*prove*(*kb*, *goal*) is used to prove *goal* from a knowledge base, *kb*, where a *goal* is a list of atoms. It returns *True* if *kb goal*. The *indent* is used when tracing the code (and doesn’t need to have 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

#### Testing:

logicTopDown.py — (continued)

29 **from** logicProblem **import** triv\_KB

30 **def** test():

31 a1 = prove(triv\_KB,['i\_am'])

32 **assert** a1, "triv\_KB proving i\_am gave "+**str**(a1)

33 a2 = prove(triv\_KB,['i\_smell'])

34 **assert not** a2, "triv\_KB proving i\_smell gave "+**str**(a2it)

35 **print**("Passed unit tests")

36 **if** name == " main ":

37 test()

38 # try

39 **from** logicProblem **import** elect

40 # elect.max\_display\_level=3 # give detailed trace

41 # prove(elect,['live\_w6'])

42 # prove(elect,['lit\_l1'])

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

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

## 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, 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)

##### Assumables 79

49 **for** ass **in** self.prove\_all\_ass(cl.body+ans\_body[1:],assumed)

50 ] # union of answers for each clause with head=selected

51 **else**: # empty body

52 **return** [assumed] # one answer

53

54 **def** conflicts(self):

55 """returns a list of minimal conflicts"""

56 **return** minsets(self.prove\_all\_ass(['false']))

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

logicAssumables.py — (continued)

58 **def** minsets(ls):

59 """ls is a list of sets

60 returns a list of minimal sets in ls

61 """

62 ans = [] # elements known to be minimal

63 **for** c **in** ls:

64 **if not any**(c1<c **for** c1 **in** ls) **and not any**(c1 <= c **for** c1 **in** ans):

65 ans.append(c)

66 **return** ans

67

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

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

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

The diagnoses can be constructed from the (minimal) conflicts as follows.

This also works if there are non-minimal conflicts, but is not as efficient.

logicAssumables.py — (continued)

69 **def** diagnoses(cons):

70 """cons is a list of (minimal) conflicts.

71 returns a list of diagnoses."""

72 **if** cons == []:

73 **return** [**set**()]

74 **else**:

75 **return** minsets([({e}|d) # | is set union

76 **for** e **in** cons[0]

77 **for** d **in** diagnoses(cons[1:])])

#### Test cases:

logicAssumables.py — (continued)

80 electa = KBA([

81 Clause('light\_l1'),

82 Clause('light\_l2'),

83 Assumable('ok\_l1'),

84 Assumable('ok\_l2'),

85 Assumable('ok\_s1'),

86 Assumable('ok\_s2'),

87 Assumable('ok\_s3'),

88 Assumable('ok\_cb1'),

89 Assumable('ok\_cb2'),

90 Assumable('live\_outside'),

91 Clause('live\_l1', ['live\_w0']),

92 Clause('live\_w0', ['up\_s2','ok\_s2','live\_w1']),

93 Clause('live\_w0', ['down\_s2','ok\_s2','live\_w2']),

94 Clause('live\_w1', ['up\_s1', 'ok\_s1', 'live\_w3']),

95 Clause('live\_w2', ['down\_s1', 'ok\_s1','live\_w3' ]),

96 Clause('live\_l2', ['live\_w4']),

97 Clause('live\_w4', ['up\_s3','ok\_s3','live\_w3' ]),

98 Clause('live\_p\_1', ['live\_w3']),

99 Clause('live\_w3', ['live\_w5', 'ok\_cb1']),

100 Clause('live\_p\_2', ['live\_w6']),

101 Clause('live\_w6', ['live\_w5', 'ok\_cb2']),

102 Clause('live\_w5', ['live\_outside']),

103 Clause('lit\_l1', ['light\_l1', 'live\_l1', 'ok\_l1']),

104 Clause('lit\_l2', ['light\_l2', 'live\_l2', 'ok\_l2']),

105 Askable('up\_s1'),

106 Askable('down\_s1'),

107 Askable('up\_s2'),

108 Askable('down\_s2'),

109 Askable('up\_s3'),

110 Askable('down\_s2'),

111 Askable('dark\_l1'),

112 Askable('dark\_l2'),

113 Clause('false', ['dark\_l1', 'lit\_l1']),

114 Clause('false', ['dark\_l2', 'lit\_l2'])

115 ])

116 # electa.prove\_all\_ass(['false'])

117 # cs=electa.conflicts()

118 # print(cs)

119 # diagnoses(cs) # diagnoses from conflicts

**Exercise 5.6** To implement a version of *conflicts* that never generates non-minimal conflicts, modify *prove all ass* to implement iterative deepening on the number of assumables used in a proof, and prune any set of assumables that is a superset of a conflict.

**Exercise 5.7** Implement *explanations*(*self* , *body*), where *body* is a list of atoms, that returns the a list of the minimal explanations of the body. This does not require modification of *prove all ass*.

**Exercise 5.8** Implement *explanations*, as in the previous question, so that it never generates non-minimal explanations. Hint: modify *prove all ass* to implement iter- ative deepening on the number of assumptions, generating conflicts and explana- tions together, and pruning as early as possible.

Chapter 6

Planning with Certainty

## Representing Actions and Planning Prob- lems

The STRIPS representation of an action consists of:

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

stripsProblem.py — STRIPS Representations of Actions

11 **class** Strips(**object**):

12 **def** init (self, preconditions, effects, cost=1):

13 """

14 defines the STRIPS represtation for an action:

15 \* preconditions is feature:value dictionary that must hold

16 for the action to be carried out

17 \* effects is a feature:value map that this action makes

18 true. The action changes the value of any feature specified

19 here, and leaves other properties unchanged.

20 \* cost is the cost of the action

21 """

22 self.preconditions = preconditions

23 self.effects = effects

24 self.cost = cost

#### A STRIPS domain consists of:

81

* A set of actions.
* A dictionary that maps each feature into a set of possible values for the feature.
* A dictionary that maps each action into a STRIPS representation of the action.

stripsProblem.py — (continued)

26 **class** STRIPS\_domain(**object**):

27 **def** init (self, feats\_vals, strips\_map):

28 """Problem domain

29 feats\_vals is a feature:domain dictionary,

30 mapping each feature to its domain

31 strips\_map is an action:strips dictionary,

32 mapping each action to its Strips representation

33 """

34 self.actions = **set**(strips\_map) # set of all actions

35 self.feats\_vals = feats\_vals

36 self.strips\_map = strips\_map

* + 1. Robot Delivery Domain

#### The following specifies the robot delivery domain of Chapter 8.

stripsProblem.py — (continued)

38 boolean = {True, False}

39 delivery\_domain = STRIPS\_domain(

40 {'RLoc':{'cs', 'off', 'lab', 'mr'}, 'RHC':boolean, 'SWC':boolean,

41 'MW':boolean, 'RHM':boolean}, #feaures:values dictionary

42 {'mc\_cs': Strips({'RLoc':'cs'}, {'RLoc':'off'}),

43 'mc\_off': Strips({'RLoc':'off'}, {'RLoc':'lab'}),

44 'mc\_lab': Strips({'RLoc':'lab'}, {'RLoc':'mr'}),

45 'mc\_mr': Strips({'RLoc':'mr'}, {'RLoc':'cs'}),

46 'mcc\_cs': Strips({'RLoc':'cs'}, {'RLoc':'mr'}),

47 'mcc\_off': Strips({'RLoc':'off'}, {'RLoc':'cs'}),

48 'mcc\_lab': Strips({'RLoc':'lab'}, {'RLoc':'off'}),

49 'mcc\_mr': Strips({'RLoc':'mr'}, {'RLoc':'lab'}),

50 'puc': Strips({'RLoc':'cs', 'RHC':False}, {'RHC':True}),

51 'dc': Strips({'RLoc':'off', 'RHC':True}, {'RHC':False, 'SWC':False}),

52 'pum': Strips({'RLoc':'mr','MW':True}, {'RHM':True,'MW':False}),

53 'dm': Strips({'RLoc':'off', 'RHM':True}, {'RHM':False})

54 } )

#### 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)

56 **class** Planning\_problem(**object**):

##### 6.1. Representing Actions and Planning Problems 83

move(b,c,a)



c

a

b

c

a

b

move(b,c,table)



b

c

a

Figure 6.1: Blocks world with two actions

57 **def** init (self, prob\_domain, initial\_state, goal):

58 """

59 a planning problem consists of

60 \* a planning domain

61 \* the initial state

62 \* a goal

63 """

64 self.prob\_domain = prob\_domain

65 self.initial\_state = initial\_state

66 self.goal = goal

67

68 problem0 = Planning\_problem(delivery\_domain,

69 {'RLoc':'lab', 'MW':True, 'SWC':True, 'RHC':False,

70 'RHM':False},

71 {'RLoc':'off'})

72 problem1 = Planning\_problem(delivery\_domain,

73 {'RLoc':'lab', 'MW':True, 'SWC':True, 'RHC':False,

74 'RHM':False},

75 {'SWC':False})

76 problem2 = Planning\_problem(delivery\_domain,

77 {'RLoc':'lab', 'MW':True, 'SWC':True, 'RHC':False,

78 'RHM':False},

79 {'SWC':False, 'MW':False, 'RHM':False})

### 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. Fig- ure [6.1](#_bookmark90) shows 3 states with some of the actions between them. The following

represents the blocks world. Note that the actions and the conditions are all strings.

stripsProblem.py — (continued)

81 ### blocks world

82 **def** move(x,y,z):

83 """string for the 'move' action"""

84 **return** 'move\_'+x+'\_from\_'+y+'\_to\_'+z

85 **def** on(x,y):

86 """string for the 'on' feature"""

87 **return** x+'\_on\_'+y

88 **def** clear(x):

89 """string for the 'clear' feature"""

90 **return** 'clear\_'+x

91 **def** create\_blocks\_world(blocks = ['a','b','c','d']):

92 blocks\_and\_table = blocks+['table']

93 stmap = {move(x,y,z):Strips({on(x,y):True, clear(x):True, clear(z):True},

94 {on(x,z):True, on(x,y):False, clear(y):True, clear(z):False})

95 **for** x **in** blocks

96 **for** y **in** blocks\_and\_table

97 **for** z **in** blocks

98 **if** x!=y **and** y!=z **and** z!=x}

99 stmap.update({move(x,y,'table'):Strips({on(x,y):True, clear(x):True},

100 {on(x,'table'):True, on(x,y):False, clear(y):True})

101 **for** x **in** blocks

102 **for** y **in** blocks

103 **if** x!=y})

104 feats\_vals = {on(x,y):boolean **for** x **in** blocks **for** y **in** blocks\_and\_table}

105 feats\_vals.update({clear(x):boolean **for** x **in** blocks\_and\_table})

106 **return** STRIPS\_domain(feats\_vals, stmap)

#### This is a classic example, with 3 blocks, and the goal consists of two conditions.

stripsProblem.py — (continued)

108 blocks1dom = create\_blocks\_world(['a','b','c'])

109 blocks1 = Planning\_problem(blocks1dom,

110 {on('a','table'):True,clear('a'):True, clear('b'):True,on('b','c'):True,

111 on('c','table'):True,clear('c'):False}, # initial state

112 {on('a','b'):True, on('c','a'):True}) #goal

#### This is a problem of inverting a tower of size 4.

stripsProblem.py — (continued)

114 blocks2dom = create\_blocks\_world(['a','b','c','d'])

115 tower4 = {clear('a'):True, on('a','b'):True, clear('b'):False,

116 on('b','c'):True, clear('c'):False, on('c','d'):True,

117 clear('b'):False, on('d','table'):True}

118 blocks2 = Planning\_problem(blocks2dom,

119 tower4, # initial state

120 {on('d','c'):True,on('c','b'):True,on('b','a'):True}) #goal

#### Moving bottom block to top of a tower of size 4.

stripsProblem.py — (continued)

122 blocks3 = Planning\_problem(blocks2dom,

123 tower4, # initial state

124 {on('d','a'):True, on('a','b'):True, on('b','c'):True}) #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** The representation of the state does not include negative *on* facts. Does it need to? 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? (Does this change an answer to the previous question?)

## Forward Planning

”stripsForwardPlanner.py”, and copy and paste the commented- out example queries at the bottom of that file.

load

in folder ”aipython”,

To run the demo,

In a forward planner, a node is a state. A state consists of an assignment, which is a variable:value dictionary. In order to be able to do multiple-path pruning, we need to define a hash function, and equality between states.

stripsForwardPlanner.py — Forward Planner with STRIPS actions

11 **from** searchProblem **import** Arc, Search\_problem

12 **from** stripsProblem **import** Strips, STRIPS\_domain

13

14 **class** State(**object**):

15 **def** init (self,assignment):

16 self.assignment = assignment

17 self.hash\_value = None

18 **def** hash (self):

19 **if** self.hash\_value **is** None:

20 self.hash\_value = **hash**(**frozenset**(self.assignment.items()))

21 **return** self.hash\_value

22 **def** eq (self,st):

23 **return** self.assignment == st.assignment

24 **def** str (self):

25 **return str**(self.assignment)

#### In order to define a search problem (page [31),](#_bookmark35) we need to define the goal condition, the start nodes, the neighbours, and (optionally) a heuristic function. Here *zero* is the default heuristic function.

stripsForwardPlanner.py — (continued)

27 **def** zero(\*args,\*\*nargs):

28 """always returns 0"""

29 **return** 0

30

31 **class** Forward\_STRIPS(Search\_problem):

32 """A search problem from a planning problem where:

33 \* a node is a state object.

34 \* the dynamics are specified by the STRIPS representation of actions

35 """

36 **def** init (self, planning\_problem, heur=zero):

37 """creates a forward seach 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 state\_asst = state.assignment

52 **return all**(prop **in** state\_asst **and** state\_asst[prop]==self.goal[prop]

53 **for** prop **in** self.goal)

54

55 **def** start\_node(self):

56 """returns start node"""

57 **return** self.initial\_state

58

59 **def** neighbors(self,state):

60 """returns neighbors of state in this problem"""

61 cost=1

62 state\_asst = state.assignment

63 **return** [ Arc(state,self.effect(act,state\_asst),cost,act)

64 **for** act **in** self.prob\_domain.actions

65 **if** self.possible(act,state\_asst)]

66

67 **def** possible(self,act,state\_asst):

68 """True if act is possible in state.

69 act is possible if all of its preconditions have the same value in the state"""

70 preconds = self.prob\_domain.strips\_map[act].preconditions

71 **return all**(pre **in** state\_asst **and** state\_asst[pre]==preconds[pre]

72 **for** pre **in** preconds)

73

74 **def** effect(self,act,state\_asst):

75 """returns the state that is the effect of doing act given state\_asst"""

76 new\_state\_asst = self.prob\_domain.strips\_map[act].effects.copy()

77 **for** prop **in** state\_asst:

78 **if** prop **not in** new\_state\_asst:

79 new\_state\_asst[prop]=state\_asst[prop]

80 **return** State(new\_state\_asst)

81

82 **def** heuristic(self,state):

83 """in the forward planner a node is a state.

84 the heuristic is an (under)estimate of the cost

85 of going from the state to the top-level goal.

86 """

87 **return** self.heur(state.assignment, self.goal)

#### Here are some test cases to try.

stripsForwardPlanner.py — (continued)

89 **from** searchBranchAndBound **import** DF\_branch\_and\_bound

90 **from** searchGeneric **import** AStarSearcher

91 **from** searchMPP **import** SearcherMPP

92 **from** stripsProblem **import** problem0, problem1, problem2, blocks1, blocks2, blocks3

93

94 # AStarSearcher(Forward\_STRIPS(problem1)).search() #A\*

95 # SearcherMPP(Forward\_STRIPS(problem1)).search() #A\* with MPP

96 # DF\_branch\_and\_bound(Forward\_STRIPS(problem1),10).search() #B&B

97 # To find more than one plan:

98 # s1 = SearcherMPP(Forward\_STRIPS(problem1)) #A\*

99 # s1.search() #find another plan

### 6.2.1 Defining Heuristics for a Planner

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

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

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

stripsHeuristic.py — Planner with Heursitic 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 it

30 if the robot needs to get coffee

31 """

32 **if** ('SWC' **in** goal **and** goal['SWC']==False

33 **and** state['SWC']==True

34 **and** state['RHC']==False):

35 **return** dist(state['RLoc'],'cs')+3

36 **else**:

37 **return** 0

#### The maximum of the values of a set of admissible heuristics is also an admis- sible heuristic. The function maxh takes a number of heuristic functions as ar- guments, 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)

#### The following runs the example with and without the heuristic. (Also try using

*AStarSearcher* instead of *SearcherMPP*.)

stripsHeuristic.py — (continued)

45 ##### Forward Planner #####

46 **from** searchGeneric **import** AStarSearcher

47 **from** searchMPP **import** SearcherMPP

48 **from** stripsForwardPlanner **import** Forward\_STRIPS

49 **from** stripsProblem **import** problem0, problem1, problem2

50

51 **def** test\_forward\_heuristic(thisproblem=problem1):

52 **print**("\n\*\*\*\*\* FORWARD NO HEURISTIC")

53 **print**(SearcherMPP(Forward\_STRIPS(thisproblem)).search())

54

55 **print**("\n\*\*\*\*\* FORWARD WITH HEURISTIC h1")

56 **print**(SearcherMPP(Forward\_STRIPS(thisproblem,h1)).search())

57

58 **print**("\n\*\*\*\*\* FORWARD WITH HEURISTICs h1 and h2")

59 **print**(SearcherMPP(Forward\_STRIPS(thisproblem,maxh(h1,h2))).search())

60

61 **if** name == " main ":

62 test\_forward\_heuristic()

**Exercise 6.4** Try the forward planner with a heuristic function of just *h*1, with just *h*2 and with both. Explain how each one prunes or doesn’t prune the search space.

**Exercise 6.5** Create a better heuristic than *maxh*(*h*1, *h*2). Try it for a number of different problems.

**Exercise 6.6** Create an admissible heuristic for the blocks world.

## Regression Planning

”stripsRegressionPlanner.py”, and copy and paste the commented- out example queries at the bottom of that file.

load

in folder ”aipython”,

To run the demo,

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

A *Subgoal* object consists of an assignment, which is *variable:value* dictionary. We make it hashable so that multiple path pruning can work. The hash is only computed when necessary (and only once).

stripsRegressionPlanner.py — Regression Planner with STRIPS actions

11 **from** searchProblem **import** Arc, Search\_problem

12

13 **class** Subgoal(**object**):

14 **def** init (self,assignment):

15 self.assignment = assignment

16 self.hash\_value = None

17 **def** hash (self):

18 **if** self.hash\_value **is** None:

19 self.hash\_value = **hash**(**frozenset**(self.assignment.items()))

20 **return** self.hash\_value

21 **def** eq (self,st):

22 **return** self.assignment == st.assignment

23 **def** str (self):

24 **return str**(self.assignment)

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

stripsRegressionPlanner.py — (continued)

26 **from** stripsForwardPlanner **import** zero

27

28 **class** Regression\_STRIPS(Search\_problem):

29 """A search problem where:

30 \* a node is a goal to be achieved, represented by a set of propositions.

31 \* the dynamics are specified by the STRIPS representation of actions

32 """

33

34 **def** init (self, planning\_problem, heur=zero):

35 """creates a regression seach 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**((g **in** self.initial\_state) **and** (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 problem"""

57 cost = 1

58 goal\_asst = subgoal.assignment

59 **return** [ Arc(subgoal,self.weakest\_precond(act,goal\_asst),cost,act)

60 **for** act **in** self.prob\_domain.actions

61 **if** self.possible(act,goal\_asst)]

62

63 **def** possible(self,act,goal\_asst):

64 """True if act is possible to achieve goal\_asst.

65

66 the action achieves an element of the effects and

67 the action doesn't delete something that needs to be achieved and

68 the precoditions are consistent with other subgoals that need to be achieved

69 """

70 effects = self.prob\_domain.strips\_map[act].effects

71 preconds = self.prob\_domain.strips\_map[act].preconditions

72 **return** ( **any**(goal\_asst[prop]==effects[prop]

73 **for** prop **in** effects **if** prop **in** goal\_asst)

74 **and all**(goal\_asst[prop]==effects[prop]

75 **for** prop **in** effects **if** prop **in** goal\_asst)

76 **and all**(goal\_asst[prop]==preconds[prop]

77 **for** prop **in** preconds **if** prop **not in** effects **and** prop **in** goal\_asst)

78 )

79

80 **def** weakest\_precond(self,act,goal\_asst):

81 """returns the subgoal that must be true so goal\_asst holds after act"""

82 new\_asst = self.prob\_domain.strips\_map[act].preconditions.copy()

83 **for** g **in** goal\_asst:

84 **if** g **not in** self.prob\_domain.strips\_map[act].effects:

85 new\_asst[g] = goal\_asst[g]

86 **return** Subgoal(new\_asst)

87

88 **def** heuristic(self,subgoal):

89 """in the regression planner a node is a subgoal.

90 the heuristic is an (under)estimate of the cost of going from the initial state to subgoal

91 """

92 **return** self.heur(self.initial\_state, subgoal.assignment)

stripsRegressionPlanner.py — (continued)

94 **from** searchBranchAndBound **import** DF\_branch\_and\_bound

95 **from** searchGeneric **import** AStarSearcher

96 **from** searchMPP **import** SearcherMPP

97 **from** stripsProblem **import** problem0, problem1, problem2

98

99 # AStarSearcher(Regression\_STRIPS(problem1)).search() #A\*

100 # SearcherMPP(Regression\_STRIPS(problem1)).search() #A\* with MPP

101 # DF\_branch\_and\_bound(Regression\_STRIPS(problem1),10).search() #B&B

**Exercise 6.7** Multiple path pruning could be used to prune more than the current code. In particular, if the current node contains more conditions than a previously visited node, it can be pruned. For example, if *a* : *True*, *b* : *False* has been visited, then any node that is a superset, e.g., *a* : *True*, *b* : *False*, *d* : *True* , need not be expanded. If the simpler subgoal does not lead to a solution, the more complicated one wont either. Implement this more severe pruning. (Hint: This may require modifications to the searcher.)

{ }

{ }

**Exercise 6.8** It is possible that, as knowledge of the domain, that some as- signment of values to variables can never be achieved. For example, the robot cannot be holding mail when there is mail waiting (assuming it isn’t holding mail initially). An assignment of values to (some of the) variables is incompat- ible if no possible (reachable) state can include that assignment. For example, j*MW*j : *True*,j *RHM*j : *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 assign- ments for the blocks world. (This should result in dramatic search improvements.)

* + 1. Defining Heuristics for a Regression Planner

The regression planner can use the same heuristic function as the forward plan- ner. 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 ex- periment with whether the same heuristic works well for both a a regression planner and a forward planner.

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

stripsHeuristic.py — (continued)

64 ##### Regression Planner

65 **from** stripsRegressionPlanner **import** Regression\_STRIPS

66

67 **def** test\_regression\_heuristic(thisproblem=problem1):

68 **print**("\n\*\*\*\*\* REGRESSION NO HEURISTIC")

69 **print**(SearcherMPP(Regression\_STRIPS(thisproblem)).search())

70

71 **print**("\n\*\*\*\*\* REGRESSION WITH HEURISTICs h1 and h2")

72 **print**(SearcherMPP(Regression\_STRIPS(thisproblem,maxh(h1,h2))).search())

73

74 **if** name == " main ":

75 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).](#_bookmark95) Explain how each one prunes or doesn’t prune the search space.

**Exercise 6.11** Create a better heuristic than *heuristic fun* defined in Section [6.2.1.](#_bookmark95)

## Planning as a CSP

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

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

This assumes the same action representation as before; we do not consider factored actions (action features), nor do we implement state constraints.

##### 6.4. Planning as a CSP 93

stripsCSPPlanner.py — CSP planner where actions are represented using STRIPS

11 **from** cspProblem **import** CSP, Constraint

12

13 **class** CSP\_from\_STRIPS(CSP):

14 """A CSP where:

15 \* a CSP variable is constructed by st(var,stage).

16 \* the dynamics are specified by the STRIPS representation of actions

17 """

18

19 **def** init (self, planning\_problem, number\_stages=2):

20 prob\_domain = planning\_problem.prob\_domain

21 initial\_state = planning\_problem.initial\_state

22 goal = planning\_problem.goal

23 self.act\_vars = [st('action',stage) **for** stage **in range**(number\_stages)]

24 domains = {av:prob\_domain.actions **for** av **in** self.act\_vars}

25 domains.update({ st(var,stage):dom

26 **for** (var,dom) **in** prob\_domain.feats\_vals.items()

27 **for** stage **in range**(number\_stages+1)})

28 # intial state constraints:

29 constraints = [Constraint((st(var,0),), is\_(val))

30 **for** (var,val) **in** initial\_state.items()]

31 # goal constraints on the final state:

32 constraints += [Constraint((st(var,number\_stages),),

33 is\_(val))

34 **for** (var,val) **in** goal.items()]

35 # precondition constraints:

36 constraints += [Constraint((st(var,stage), st('action',stage)),

37 if\_(val,act)) # st(var,stage)==val if st('action',stage)=act

38 **for** act,strps **in** prob\_domain.strips\_map.items()

39 **for** var,val **in** strps.preconditions.items()

40 **for** stage **in range**(number\_stages)]

41 # effect constraints:

42 constraints += [Constraint((st(var,stage+1), st('action',stage)),

43 if\_(val,act)) # st(var,stage+1)==val if st('action',stage)==act

44 **for** act,strps **in** prob\_domain.strips\_map.items()

45 **for** var,val **in** strps.effects.items()

46 **for** stage **in range**(number\_stages)]

47 # frame constraints:

48 constraints += [Constraint((st(var,stage), st('action',stage), st(var,stage+1)),

49 eq\_if\_not\_in\_({act **for** act **in** prob\_domain.actions

50 **if** var **in** prob\_domain.strips\_map[act].effects}))

51 **for** var **in** prob\_domain.feats\_vals

52 **for** stage **in range**(number\_stages) ]

53 CSP. init (self, domains, constraints)

54

55 **def** extract\_plan(self,soln):

56 **return** [soln[a] **for** a **in** self.act\_vars]

57

58 **def** st(var,stage):

59 """returns a string for the var-stage pair that can be used as a variable"""

60 **return str**(var)+"\_"+**str**(stage)

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

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

Note that the underscore (’ ’) is part of the name; here we use it as the

convention that it is a function that returns a function. This uses two different styles to define *is* and *if* ; returning a function defined by *lambda* is equivaent 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)

62 **def** is\_(val):

63 """returns a function that is true when it is it applied to val.

64 """

65 **return lambda** x: x == val

66

67 **def** if\_(v1,v2):

68 """if the second argument is v2, the first argument must be v1"""

69 #return lambda x1,x2: x1==v1 if x2==v2 else True

70 **def** if\_fun(x1,x2):

71 **return** x1==v1 **if** x2==v2 **else** True

72 if\_fun. doc = "if x2 is "+**str**(v2)+" then x1 is "+**str**(v1)

73 **return** if\_fun

74

75 **def** eq\_if\_not\_in\_(actset):

76 """first and third arguments are equal if action is not in actset"""

77 **return lambda** x1, a, x2: x1==x2 **if** a **not in** actset **else** True

#### Putting it together, this returns a list of actions that solves the problem *prob* for a given horizon. If you want to do more than just return the list of actions, you might want to get it to return the solution. Or even enumerate the solutions (by using *Search with AC from CSP*).

stripsCSPPlanner.py — (continued)

79 **def** con\_plan(prob,horizon):

80 """finds a plan for problem prob given horizon.

81 """

82 csp = CSP\_from\_STRIPS(prob, horizon)

83 sol = Con\_solver(csp).solve\_one()

84 **return** csp.extract\_plan(sol) **if** sol **else** sol

#### The following are some example queries.

stripsCSPPlanner.py — (continued)

86 **from** searchGeneric **import** Searcher

87 **from** stripsProblem **import** delivery\_domain

88 **from** cspConsistency **import** Search\_with\_AC\_from\_CSP, Con\_solver

89 **from** stripsProblem **import** Planning\_problem, problem0, problem1, problem2

90

91 # Problem 0

92 # con\_plan(problem0,1) # should it succeed?

93 # con\_plan(problem0,2) # should it succeed?

94 # con\_plan(problem0,3) # should it succeed?

95 # To use search to enumerate solutions

96 #searcher0a = Searcher(Search\_with\_AC\_from\_CSP(CSP\_from\_STRIPS(problem0, 1)))

97 #print(searcher0a.search())

98

99 ## Problem 1

100 # con\_plan(problem1,5) # should it succeed?

101 # con\_plan(problem1,4) # should it succeed?

102 ## To use search to enumerate solutions:

103 #searcher15a = Searcher(Search\_with\_AC\_from\_CSP(CSP\_from\_STRIPS(problem1, 5)))

104 #print(searcher15a.search())

105

106 ## Problem 2

107 #con\_plan(problem2, 6) # should fail??

108 #con\_plan(problem2, 7) # should succeed???

109

110 ## Example 6.13

111 problem3 = Planning\_problem(delivery\_domain,

112 {'SWC':True, 'RHC':False}, {'SWC':False})

113 #con\_plan(problem3,2) # Horizon of 2

114 #con\_plan(problem3,3) # Horizon of 3

115

116 problem4 = Planning\_problem(delivery\_domain,{'SWC':True},

117 {'SWC':False, 'MW':False, 'RHM':False})

118

119 # For the stochastic local search:

120 #from cspSLS import SLSearcher, Runtime\_distribution

121 # cspplanning15 = CSP\_from\_STRIPS(problem1, 5) # should succeed

122 #se0 = SLSearcher(cspplanning15); print(se0.search(100000,0.5))

123 #p = Runtime\_distribution(cspplanning15)

124 #p.plot\_run(1000,1000,0.7) # warning will take a few minutes

## 6.5 Partial-Order Planning

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

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

stripsPOP.py — Partial-order Planner using STRIPS representation

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 str**(self.action)+"#"+**str**(self.index)

25

26 repr = str # repr function is the same as the str function

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

* + - * *actions*: a set of action instances.
      * *constraints*: a set of (*a*1, *a*2) pairs, where *a*1 and *a*2 are action instances, which represents that *a*1 must come before *a*2 in the partial order. There are a number of ways that this could be represented. Here we represent the set of pairs that are in transitive closure of the *before* relation. This lets us quickly determine whether some before relation is consistent with the current constraints.
      * *agenda*: a list of (*s*, *a*) pairs, where *s* is a (*var*, *val*) pair and *a* is an action instance. This means that variable *var* must have value *val* before *a* can occur.
      * *causal links*: a set of (*a*0, *g*, *a*1) triples, where *a*1 and *a*2 are action instances and *g* is a (*var*, *val*) pair. This holds when action *a*0 makes *g* true for action *a*1.

stripsPOP.py — (continued)

28 **class** POP\_node(**object**):

29 """a (partial) partial-order plan. This is a node in the search space."""

30 **def** init (self, actions, constraints, agenda, causal\_links):

31 """

32 \* actions is a set of action instances

33 \* constraints a set of (a0,a1) pairs, representing a0<a1,

34 closed under transitivity

35 \* agenda list of (subgoal,action) pairs to be achieved, where

36 subgoal is a (variable,value) pair

37 \* causal\_links is a set of (a0,g,a1) triples,

38 where ai are action instances, and g is a (variable,value) pair

39 """

40 self.actions = actions # a set of action instances

41 self.constraints = constraints # a set of (a0,a1) pairs

42 self.agenda = agenda # list of (subgoal,action) pairs to be achieved

43 self.causal\_links = causal\_links # set of (a0,g,a1) triples

44

45 **def** str (self):

46 **return** ("actions: "+**str**({**str**(a) **for** a **in** self.actions})+

47 "\nconstraints: "+

48 **str**({(**str**(a1),**str**(a2)) **for** (a1,a2) **in** self.constraints})+

49 "\nagenda: "+

50 **str**([(**str**(s),**str**(a)) **for** (s,a) **in** self.agenda])+

51 "\ncausal\_links:"+

52 **str**({(**str**(a0),**str**(g),**str**(a2)) **for** (a0,g,a2) **in** self.causal\_links})

)

#### *extract plan* constructs a total order of action instances that is consistent with

the partial order.

stripsPOP.py — (continued)

54 **def** extract\_plan(self):

55 """returns a total ordering of the action instances consistent

56 with the constraints.

57 raises IndexError if there is no choice.

58 """

59 sorted\_acts = []

60 other\_acts = **set**(self.actions)

61 **while** other\_acts:

62 a = random.choice([a **for** a **in** other\_acts **if**

63 **all**(((a1,a) **not in** self.constraints) **for** a1 **in** other\_acts)])

64 sorted\_acts.append(a)

65 other\_acts.remove(a)

66 **return** sorted\_acts

#### *POP search from STRIPS* is an instance of a search problem. As such, we need to define the start nodes, the goal, and the neighbors of a node.

stripsPOP.py — (continued)

68 **from** display **import** Displayable

69

70 **class** POP\_search\_from\_STRIPS(Search\_problem, Displayable):

71 **def** init (self,planning\_problem):

72 Search\_problem. init (self)

73 self.planning\_problem = planning\_problem

74 self.start = Action\_instance("start")

75 self.finish = Action\_instance("finish")

76

77 **def** is\_goal(self, node):

78 **return** node.agenda == []

79

80 **def** start\_node(self):

81 constraints = {(self.start, self.finish)}

82 agenda = [(g, self.finish) **for** g **in** self.planning\_problem.goal.items()]

83 **return** POP\_node([self.start,self.finish], constraints, agenda, [] )

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

stripsPOP.py — (continued)

85 **def** neighbors(self, node):

86 """enumerates the neighbors of node"""

87 self.display(3,"finding neighbors of\n",node)

88 **if** node.agenda:

89 subgoal,act1 = node.agenda[0]

90 self.display(2,"selecting",subgoal,"for",act1)

91 new\_agenda = node.agenda[1:]

92 **for** act0 **in** node.actions:

93 **if** (self.achieves(act0, subgoal) **and**

94 self.possible((act0,act1),node.constraints)):

95 self.display(2," reusing",act0)

96 consts1 = self.add\_constraint((act0,act1),node.constraints)

97 new\_clink = (act0,subgoal,act1)

98 new\_cls = node.causal\_links + [new\_clink]

99 **for** consts2 **in** self.protect\_cl\_for\_actions(node.actions,consts1,new\_clink):

100 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.strips\_map: #a0 is an action

104 **if** self.achieves(a0, subgoal):

105 #a0 acheieves 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 preconds = self.planning\_problem.prob\_domain.strips\_map[a0].preconditions

112 new\_agenda = new\_agenda + [(pre,new\_a) **for** pre **in** preconds.items()]

113 new\_clink = (new\_a,subgoal,act1)

114 new\_cls = node.causal\_links + [new\_clink]

115 **for** consts3 **in** self.protect\_all\_cls(node.causal\_links,new\_a,consts2):

116 **for** consts4 **in** self.protect\_cl\_for\_actions(node.actions,consts3,new\_clink):

117 yield Arc(node,

118 POP\_node(new\_actions,consts4,new\_agenda,new\_cls),

119 cost=1)

#### Given a casual link (*a*0, *subgoal*, *a*1), the following method protects the causal link from each action in *actions*. Whenever an action deletes *subgoal*, the action needs to be before *a*0 or after *a*1. This method enumerates all constraints that result from protecting the causal link from all actions.

stripsPOP.py — (continued)

121 **def** protect\_cl\_for\_actions(self, actions, constrs, clink):

122 """yields constriants that extend constrs and

123 protect causal link (a0, subgoal, a1)

124 for each action in actions

125 """

126 **if** actions:

127 a = actions[0]

128 rem\_actions = actions[1:]

129 a0, subgoal, a1 = clink

130 **if** a != a0 **and** a != a1 **and** self.deletes(a,subgoal):

131 **if** self.possible((a,a0),constrs):

132 new\_const = self.add\_constraint((a,a0),constrs)

133 **for** e **in** self.protect\_cl\_for\_actions(rem\_actions,new\_const,clink): yield e # could be "yield from"

134 **if** self.possible((a1,a),constrs):

135 new\_const = self.add\_constraint((a1,a),constrs)

136 **for** e **in** self.protect\_cl\_for\_actions(rem\_actions,new\_const,clink): yield e

137 **else**:

138 **for** e **in** self.protect\_cl\_for\_actions(rem\_actions,constrs,clink): yield e

139 **else**:

140 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 (*a*0, *subgoal*, *a*1), the action *act* needs to be before *a*0 or after *a*1. This method enumerates all con- straints that result from protecting the causal links from *act*.

stripsPOP.py — (continued)

142 **def** protect\_all\_cls(self, clinks, act, constrs):

143 """yields constraints that protect all causal links from act"""

144 **if** clinks:

145 (a0,cond,a1) = clinks[0] # select a causal link

146 rem\_clinks = clinks[1:] # remaining causal links

147 **if** act != a0 **and** act != a1 **and** self.deletes(act,cond):

148 **if** self.possible((act,a0),constrs):

149 new\_const = self.add\_constraint((act,a0),constrs)

150 **for** e **in** self.protect\_all\_cls(rem\_clinks,act,new\_const): 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): yield e

154 **else**:

155 **for** e **in** self.protect\_all\_cls(rem\_clinks,act,constrs): yield e

156 **else**:

157 yield constrs

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

stripsPOP.py — (continued)

159 **def** achieves(self,action,subgoal):

160 var,val = subgoal

161 **return** var **in** self.effects(action) **and** self.effects(action)[var] == val

162

163 **def** deletes(self,action,subgoal):

164 var,val = subgoal

165 **return** var **in** self.effects(action) **and** self.effects(action)[var] != val

166

167 **def** effects(self,action):

168 """returns the variable:value dictionary of the effects of action.

169 works for both actions and action instances"""

170 **if isinstance**(action, Action\_instance):

171 action = action.action

172 **if** action == "start":

173 **return** self.planning\_problem.initial\_state

174 **elif** action == "finish":

175 **return** {}

176 **else**:

177 **return** self.planning\_problem.prob\_domain.strips\_map[action].effects

#### The constriants 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)

179 **def** add\_constraint(self, pair, const):

180 **if** pair **in** const:

181 **return** const

182 todo = [pair]

183 newconst = const.copy()

184 **while** todo:

185 x0,x1 = todo.pop()

186 newconst.add((x0,x1))

187 **for** x,y **in** newconst:

188 **if** x==x1 **and** (x0,y) **not in** newconst:

189 todo.append((x0,y))

190 **if** y==x0 **and** (x,x1) **not in** newconst:

191 todo.append((x,x1))

192 **return** newconst

193

194 **def** possible(self,pair,constraint):

195 (x,y) = pair

196 **return** (y,x) **not in** constraint

#### Some code for testing:

stripsPOP.py — (continued)

198 **from** searchBranchAndBound **import** DF\_branch\_and\_bound

199 **from** searchGeneric **import** AStarSearcher

200 **from** searchMPP **import** SearcherMPP

201 **from** stripsProblem **import** problem0, problem1, problem2

202

203 rplanning0 = POP\_search\_from\_STRIPS(problem0)

204 rplanning1 = POP\_search\_from\_STRIPS(problem1)

205 rplanning2 = POP\_search\_from\_STRIPS(problem2)

206 searcher0 = DF\_branch\_and\_bound(rplanning0,5)

207 searcher0a = AStarSearcher(rplanning0)

|  |  |  |
| --- | --- | --- |
| 208  209  210  211 | searcher1 = DF\_branch\_and\_bound(rplanning1,10)  searcher1a = AStarSearcher(rplanning1) searcher2 = DF\_branch\_and\_bound(rplanning2,10) searcher2a = AStarSearcher(rplanning2) | |
| 212 | # | Try one of the following searchers |
| 213 | # | a = searcher0.search() |
| 214 | # | a = searcher0a.search() |
| 215 | # | a.end().extract\_plan() # print a plan found |
| 216 | # | a.end().constraints # print the constraints |
| 217 | # | AStarSearcher.max\_display\_level = 0 # less detailed display |
| 218 | # | DF\_branch\_and\_bound.max\_display\_level = 0 # less detailed display |
| 219 | # | a = searcher1.search() |
| 220 | # | a = searcher1a.search() |
| 221 | # | a = searcher2.search() |
| 222 | # | a = searcher2a.search() |

# Chapter 7

Supervised Machine Learning

#### A good source of datasets is the UCI machine Learning Repository [**?**]; the SPECT and car datasets are from this repository.

## Representations of Data and Predictions

The code uses the following deinitions and conventions:

* + - A **data set** is an enumeration of examples.

#### An **example** is a list (or tuple) of feature values. The feature values can be numbers or strings.

* + - A **feature** is a function from examples into the range of the feature. We assume each feature has a variable frange that gives the range of the fea- ture.

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

{ }

#### The doc variable of the function contains the docstring, a string de- scription of the function.

learnProblem.py — A Learning Problem

11 **import** math, random

12 **import** csv

13 **from** display **import** Displayable

14

15 boolean = [False, True]

#### 103

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

learnProblem.py — (continued)

17 **class** Data\_set(Displayable):

18 """ A data set consists of a list of training data and a list of test data.

19 """

20 seed = None #123456 # make it None for a different test set each time

21

22 **def** init (self, train, test=None, prob\_test=0.30, target\_index=0, header=None):

23 """A dataset for learning.

24 train is a list of tuples representing the training examples

25 test is the list of tuples representing the test examples

26 if test is None, a test set is created by selecting each

27 example with probability prob\_test

28 target\_index is the index of the target. If negative, it counts from right.

29 If target\_index is larger than the number of properties,

30 there is no target (for unsupervised learning)

31 header is a list of names for the features

32 """

33 **if** test **is** None:

34 train,test = partition\_data(train, prob\_test, seed=self.seed)

35 self.train = train

36 self.test = test

37 self.display(1,"Tuples read. \nTraining set", **len**(train),

38 "examples. Number of columns:",{**len**(e) **for** e **in** train},

39 "\nTest set", **len**(test),

40 "examples. Number of columns:",{**len**(e) **for** e **in** test}

41 )

42 self.prob\_test = prob\_test

43 self.num\_properties = **len**(self.train[0])

44 **if** target\_index < 0: #allows for -1, -2, etc.

45 target\_index = self.num\_properties + target\_index

46 self.target\_index = target\_index

47 self.header = header

48 self.create\_features()

49 self.display(1,"There are",**len**(self.input\_features),"input features")

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

learnProblem.py — (continued)

51 **def** create\_features(self):

52 """create the input features and target feature.

53 This assumes that the features all have range {0,1}.

54 This should be overridden if the features have a different range.

55 """

56 self.input\_features = []

57 **for** i **in range**(self.num\_properties):

58 **def** feat(e,index=i):

59 **return** e[index]

60 **if** self.header:

61 feat. doc = self.header[i]

62 **else**:

63 feat. doc = "e["+**str**(i)+"]"

64 feat.frange = [0,1]

65 **if** i == self.target\_index:

66 self.target = feat

67 **else**:

68 self.input\_features.append(feat)

* + 1. Evaluating Predictions

#### A **predictor** is a function that takes an example and makes a prediction on the value of the target feature. A predictor can be judged according to a number of evaluation criteria. The function *evaluate dataset* returns the average error for each example, where the error for each example depends on the evaluation criteria. Here we consider three evaluation criteria, the sum-of-squares, the sum of absolute errors and the logloss (the negative log-likelihood, which is the number of bits to describe the data using a code based on the prediction treated as a probability).

learnProblem.py — (continued)

70 evaluation\_criteria = ["sum-of-squares","sum\_absolute","logloss"]

71

72 **def** evaluate\_dataset(self, data, predictor, evaluation\_criterion):

73 """Evaluates predictor on data according to the evaluation\_criterion.

74 predictor is a function that takes an example and returns a

75 prediction for the target feature.

76 evaluation\_criterion is one of the evaluation\_criteria.

77 """

78 **assert** evaluation\_criterion **in** self.evaluation\_criteria,"given: "+**str**(evaluation\_criterion

79 **if** data:

80 **try**:

81 error = **sum**(error\_example(predictor(example), self.target(example),

82 evaluation\_criterion)

83 **for** example **in** data)/**len**(data)

84 **except** ValueError:

85 **return float**("inf") # infinity

86 **return** error

#### *error example* is used to evaluate a single example, based on the predicted value, the actual value and the evaluation criterion. Note that for logloss, the actual value must be 0 or 1.

learnProblem.py — (continued)

88 **def** error\_example(predicted, actual, evaluation\_criterion):

89 """returns the error of the for the predicted value given the actual value

90 according to evaluation\_criterion.

91 Throws ValueError if the error is infinite (log(0))

92 """

93 **if** evaluation\_criterion=="sum-of-squares":

94 **return** (predicted-actual)\*\*2

95 **elif** evaluation\_criterion=="sum\_absolute":

96 **return abs**(predicted-actual)

97 **elif** evaluation\_criterion=="logloss":

98 **assert** actual **in** [0,1], "actual="+**str**(actual)

99 **if** actual==0:

100 **return** -math.log2(1-predicted)

101 **else**:

102 **return** -math.log2(predicted)

103 **elif** evaluation\_criterion=="characteristic\_ss":

104 **return sum**((1-predicted[i])\*\*2 **if** actual==i **else** predicted[i]\*\*2

105 **for** i **in range**(**len**(predicted)))

106 **else**:

107 **raise** RuntimeError("Not evaluation criteria: "+**str**(evaluation\_criterion))

### Creating Test and Training Sets

#### The following method partitions the data into a training set and a test set. Note that this does not guarantee that the test set will contain exactly a proportion of the data equal to *prob test*.

[An alternative is to use *random*.*sample*() which can guarantee that the test set will contain exactly a particular proportion of the data. However this would require knowing how many elements are in the data set, which we may not know, as *data* may just be a generator of the data (e.g., when reading the data from a file).]

learnProblem.py — (continued)

109 **def** partition\_data(data, prob\_test=0.30, seed=None):

110 """partitions the data into a training set and a test set, where

111 prob\_test is the probability of each example being in the test set.

112 """

113 train = []

114 test = []

115 **if** seed: # given seed makes the partition consistent from run-to-run

116 random.seed(seed)

117 **for** example **in** data:

118 **if** random.random() < prob\_test:

119 test.append(example)

120 **else**:

121 train.append(example)

122 **return** train, test

### Importing Data From File

#### A data set is typically loaded from a file. The default here is that it loaded from a CSV (comma separated values) file, although the default separator can be changed. This assumes that all lines that contain the separator are valid data (so we only include those data items that contain more than one element). This allows for blank lines and comment lines that do not contain the separator. However, it means that this method is not suitable for cases where there is only one feature.

Note that *data all* and *data tuples* are generators. *data all* is a generator of a list of list of strings. This version assumes that CSV files are simple. The standard *csv* package, that allows quoted arguments, can be used by uncom- menting the line for *data aa* and commenting out the following line. *data tuples* contains only those lines that contain the delimiter (others lines are assumed to be empty or comments), and tries to convert the elements to numbers when- ever possible.

This allows for some of the columns to be included. Note that if include only is specified, the target index is in the resulting

learnProblem.py — (continued)

124 **class** Data\_from\_file(Data\_set):

125 **def** init (self, file\_name, separator=',', num\_train=None, prob\_test=0.3,

126 has\_header=False, target\_index=0, boolean\_features=True,

127 categorical=[], include\_only=None):

128 """create a dataset from a file

129 separator is the character that separates the attributes

130 num\_train is a number n specifying the first n tuples are training, or None

131 prob\_test is the probability an example should in the test set (if num\_train is None)

132 has\_header is True if the first line of file is a header

133 target\_index specifies which feature is the target

134 boolean\_features specifies whether we want to create Boolean features

135 (if False, is uses the original features).

136 categorical is a set (or list) of features that should be treated as categorical

137 include\_only is a list or set of indexes of columns to include

138 """

139 self.boolean\_features = boolean\_features

140 with **open**(file\_name,'r',newline='') as csvfile:

141 # data\_all = csv.reader(csvfile,delimiter=separator) # for more complicted CSV files

142 data\_all = (line.strip().split(separator) **for** line **in** csvfile)

143 **if** include\_only **is not** None:

144 data\_all = ([v **for** (i,v) **in enumerate**(line) **if** i **in** include\_only] **for** line **in** data\_

145 **if** has\_header:

146 header = **next**(data\_all)

147 **else**:

148 header = None

149 data\_tuples = (make\_num(d) **for** d **in** data\_all **if len**(d)>1)

150 **if** num\_train **is not** None:

151 # training set is divided into training then text examples

152 # the file is only read once, and the data is placed in appropriate list

|  |  |
| --- | --- |
| 153 | train = [] |
| 154 | **for** i **in range**(num\_train): # will give an error if insufficient examples |
| 155 | train.append(**next**(data\_tuples)) |
| 156 | test = **list**(data\_tuples) |
| 157 | Data\_set. init (self,train, test=test, target\_index=target\_index,header=header) |
| 158 | **else**: # randomly assign training and test examples |
| 159 | Data\_set. init (self,data\_tuples, prob\_test=prob\_test, |
| 160 | target\_index=target\_index, header=header) |
| 161 |  |
| 162 | **def** str (self): |
| 163 | **if** self.train **and len**(self.train)>0: |
| 164 | **return** ("Data: "+**str**(**len**(self.train))+" training examples, " |
| 165 | +**str**(**len**(self.test))+" test examples, " |
| 166 | +**str**(**len**(self.train[0]))+" features.") |
| 167 | **else**: |
| 168 | **return** ("Data: "+**str**(**len**(self.train))+" training examples, " |
| 169 | +**str**(**len**(self.test))+" test examples.") |

### Creating Binary Features

#### Some of the algorithms require Boolean features or features with range 0, 1 . In order to be able to use these algorithms on datasets that allow for arbitrary ranges of input variables, we construct binary features from the attributes. This method overrides the method in *Data set*.

{ }

There are 3 cases:

* + - * When the range only has two values, we designate one to be the “true” value.
      * When the values are all numeric, we assume they are ordered (as opposed to just being some classes that happen to be labelled with numbers, but where the numbers have no meaning) and construct Boolean features for splits of the data. That is, the feature is *e*[*ind*] *< cut* for some value *cut*. We choose a number of *cut* values, up to a maximum number of cuts, given by *max num cuts*.
      * When the values are not all numeric, we assume they are unordered, and create an indicator function for each value. An indicator function for a value returns true when that value is given and false otherwise. Note that we can’t create an indicator function for values that appear in the test set but not in the training set because we haven’t seen the test set. For the examples in the test set with that value, the indicator functions all return false.

learnProblem.py — (continued)

171 **def** create\_features(self, max\_num\_cuts=8):

172 """creates boolean features from input features.

|  |  |
| --- | --- |
| 173 | max\_num\_cuts is the maximum number of binary variables |
| 174 | to split a numerical feature into. |
| 175 | """ |
| 176 | ranges = [**set**() **for** i **in range**(self.num\_properties)] |
| 177 | **for** example **in** self.train: |
| 178 | **for** ind,val **in enumerate**(example): |
| 179 | ranges[ind].add(val) |
| 180 | **if** self.target\_index <= self.num\_properties: |
| 181 | **def** target(e,index=self.target\_index): |
| 182 | **return** e[index] |
| 183 | **if** self.header: |
| 184 | target. doc = self.header[ind] |
| 185 | **else**: |
| 186 | target. doc = "e["+**str**(ind)+"]" |
| 187 | target.frange = ranges[self.target\_index] |
| 188 | self.target = target |
| 189 | **if** self.boolean\_features: |
| 190 | self.input\_features = [] |
| 191 | **for** ind,frange **in enumerate**(ranges): |
| 192 | **if** ind != self.target\_index **and len**(frange)>1: |
| 193 | **if len**(frange) == 2: |
| 194 | # two values, the feature is equality to one of them. |
| 195 | true\_val = **list**(frange)[1] # choose one as true |
| 196 | **def** feat(e, i=ind, tv=true\_val): |
| 197 | **return** e[i]==tv |
| 198 | **if** self.header: |
| 199 | feat. doc = self.header[ind]+"=="+**str**(true\_val) |
| 200 | **else**: |
| 201 | feat. doc = "e["+**str**(ind)+"]=="+**str**(true\_val) |
| 202 | feat.frange = boolean |
| 203 | self.input\_features.append(feat) |
| 204 | **elif all**(**isinstance**(val,(**int**,**float**)) **for** val **in** frange): |
| 205 | # all numeric, create cuts of the data |
| 206 | sorted\_frange = **sorted**(frange) |
| 207 | num\_cuts = **min**(max\_num\_cuts,**len**(frange)) |
| 208 | cut\_positions = [**len**(frange)\*i//num\_cuts **for** i **in range**(1,num\_cuts)] |
| 209 | **for** cut **in** cut\_positions: |
| 210 | cutat = sorted\_frange[cut] |
| 211 | **def** feat(e, ind\_=ind, cutat=cutat): |
| 212 | **return** e[ind\_] < cutat |
| 213 |  |
| 214 | **if** self.header: |
| 215 | feat. doc = self.header[ind]+"<"+**str**(cutat) |
| 216 | **else**: |
| 217 | feat. doc = "e["+**str**(ind)+"]<"+**str**(cutat) |
| 218 | feat.frange = boolean |
| 219 | self.input\_features.append(feat) |
| 220 | **else**: |
| 221 | # create an indicator function for every value |
| 222 | **for** val **in** frange: |

223 **def** feat(e, ind\_=ind, val\_=val):

224 **return** e[ind\_] == val\_

225 **if** self.header:

226 feat. doc = self.header[ind]+"=="+**str**(val)

227 **else**:

228 feat. doc = "e["+**str**(ind)+"]=="+**str**(val)

229 feat.frange = boolean

230 self.input\_features.append(feat)

231 **else**: # boolean\_features is off

232 self.input\_features = []

233 **for** i **in range**(self.num\_properties):

234 **def** feat(e,index=i):

235 **return** e[index]

236 **if** self.header:

237 feat. doc = self.header[i]

238 **else**:

239 feat. doc = "e["+**str**(i)+"]"

240 feat.frange = ranges[i]

241 **if** i == self.target\_index:

242 self.target = feat

243 **else**:

244 self.input\_features.append(feat)

**Exercise 7.1** Change the code so that it splits using *e*[*ind*] *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*] 109 and *e*[*ind*] 119 to make sure that each of the resulting ranges is equal size.

≤

≤ ≤

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

*cutat* = (*sorted frange*[*cut*] + *sorted frange*[*cut* − 1])/2

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

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

learnProblem.py — (continued)

245 **def** make\_num(str\_list):

246 """make the elements of string list str\_list numerical if possible.

247 Otherwise remove initial and trailing spaces.

248 """

249 res = []

250 **for** e **in** str\_list:

251 **try**:

252 res.append(**int**(e))

253 **except** ValueError:

254 **try**:

255 res.append(**float**(e))

256 **except** ValueError:

257 res.append(e.strip())

258 **return** res

### Augmented Features

#### Sometimes we want to augment the features with new features computed from the old features (eg. the product of features). Here we allow the creation of a new dataset from an old dataset but with new features.

A feature is a function of examples. A unary feature constructor takes a fea- ture and returns a new feature. A binary feature combiner takes two features and returns a new feature.

learnProblem.py — (continued)

260 **class** Data\_set\_augmented(Data\_set):

261 **def** init (self, dataset, unary\_functions=[], binary\_functions=[], include\_orig=True):

262 """creates a dataset like dataset but with new features

263 unary\_function is a list of unary feature constructors

264 binary\_functions is a list of binary feature combiners.

265 include\_orig specifies whether the original features should be included

266 """

267 self.orig\_dataset = dataset

268 self.unary\_functions = unary\_functions

269 self.binary\_functions = binary\_functions

270 self.include\_orig = include\_orig

271 self.target = dataset.target

272 Data\_set. init (self,dataset.train, test=dataset.test,

273 target\_index = dataset.target\_index)

274

275 **def** create\_features(self):

276 **if** self.include\_orig:

277 self.input\_features = self.orig\_dataset.input\_features.copy()

278 **else**:

279 self.input\_features = []

280 **for** u **in** self.unary\_functions:

281 **for** f **in** self.orig\_dataset.input\_features:

282 self.input\_features.append(u(f))

283 **for** b **in** self.binary\_functions:

284 **for** f1 **in** self.orig\_dataset.input\_features:

285 **for** f2 **in** self.orig\_dataset.input\_features:

286 **if** f1 != f2:

287 self.input\_features.append(b(f1,f2))

#### The following are useful unary feature constructors and binary feature com- biner.

learnProblem.py — (continued)

289 **def** square(f):

|  |  |  |
| --- | --- | --- |
| 290 |  | """a unary feature constructor to construct the square of a feature |
| 291 |  | """ |
| 292 |  | **def** sq(e): |
| 293 |  | **return** f(e)\*\*2 |
| 294 |  | sq. doc = f. doc +"\*\*2" |
| 295 |  | **return** sq |
| 296 |  |  |
| 297 | **def** | power\_feat(n): |
| 298 |  | """given n returns a unary feature constructor to construct the nth power of a feature. |
| 299 |  | e.g., power\_feat(2) is the same as square |
| 300 |  | """ |
| 301 |  | **def** fn(f,n=n): |
| 302 |  | **def pow**(e,n=n): |
| 303 |  | **return** f(e)\*\*n |
| 304 |  | **pow**. doc = f. doc +"\*\*"+**str**(n) |
| 305 |  | **return pow** |
| 306 |  | **return** fn |
| 307 |  |  |
| 308 | **def** | prod\_feat(f1,f2): |
| 309 |  | """a new feature that is the product of features f1 and f2 |
| 310 |  | """ |
| 311 |  | **def** feat(e): |
| 312 |  | **return** f1(e)\*f2(e) |
| 313 |  | feat. doc = f1. doc +"\*"+f2. doc |
| 314 |  | **return** feat |
| 315 |  |  |
| 316 | **def** | eq\_feat(f1,f2): |
| 317 |  | """a new feature that is 1 if f1 and f2 give same value |
| 318 |  | """ |
| 319 |  | **def** feat(e): |
| 320 |  | **return** 1 **if** f1(e)==f2(e) **else** 0 |
| 321 |  | feat. doc = f1. doc +"=="+f2. doc |
| 322 |  | **return** feat |
| 323 |  |  |
| 324 | **def** | neq\_feat(f1,f2): |
| 325 |  | """a new feature that is 1 if f1 and f2 give different values |
| 326 |  | """ |
| 327 |  | **def** feat(e): |
| 328 |  | **return** 1 **if** f1(e)!=f2(e) **else** 0 |
| 329 |  | feat. doc = f1. doc +"!="+f2. doc |
| 330 |  | **return** feat |

#### Example:

learnProblem.py — (continued)

332 # from learnProblem import Data\_set\_augmented,prod\_feat

333 # data = Data\_from\_file('data/holiday.csv', num\_train=19, target\_index=-1)

334 ## data = Data\_from\_file('data/SPECT.csv', prob\_test=0.5, target\_index=0)

335 # dataplus = Data\_set\_augmented(data,[],[prod\_feat])

336 # 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.

∗ ∗

* + 1. Learner

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

|  |  |  |
| --- | --- | --- |
| 337  338  339  340  341 |  | learnProblem.py — (continued)  **from** display **import** Displayable  **class** Learner(Displayable):  **def** init (self, dataset):  **raise** NotImplementedError("Learner. init ") # abstract method |
| 342 |  |  |
| 343 |  | **def** learn(self): |
| 344 |  | """returns a predictor, a function from a tuple to a value for the target feature |
| 345 |  | """ |
| 346 |  | **raise** NotImplementedError("learn") # abstract method |

## Learning With No Input Features

If we make the same prediction for each example, what prediction should we make?

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 exam- ple, if the values of the feature are 0, 1 we may be allowed to predict 0.3, 1, or even 1.7. For all of the criteria we can imagine, there is no point in predicting a value greater than 1 or less that zero (but that doesn’t mean we can’t), but it is often useful to predict a value between 0 and 1. If the values are ratings in {1, 2, 3, 4, 5}, we may want to predict 3.4.

{ }

* + - a probability distribution over the values of the feature. For each value *v*, we predict a non-negative number *pv*, such that the sum over all predic- tions is 1.

The following code assumes the second of these, where we can make a point prediction of any value (although median will only predict one of the actual values for the feature).

The *point prediction* function takes in a target feature (which is assumed to be numeric), some training data, and a section of what to return, and returns a function that takes in an example, and makes a prediction of a value for the target variable, but makes same prediction for all examples.

This method uses *selection*, whose value should be “median”, “proportion”, or “Laplace” determine what prediction should be made.

learnNoInputs.py — Learning ignoring all input features

11 **from** learnProblem **import** Learner, Data\_set

12 **import** math, random

13

14 selections = ["median", "mean", "Laplace"]

15

16 **def** point\_prediction(target, training\_data,

17 selection="mean" ):

18 """makes a point prediction for a set of training data.

19 target provides the target

20 training\_data provides the training data to use (often a subset of train).

21 selection specifies what statistic of the data to use as the evaluation.

22 to\_optimize provides a criteria to optimize (used to guess selection)

23 """

24 **assert len**(training\_data)>0

25 **if** selection == "median":

26 counts,total = target\_counts(target,training\_data)

27 middle = total/2

28 cumulative = 0

29 **for** val,num **in sorted**(counts.items()):

30 cumulative += num

31 **if** cumulative > middle:

32 **break** # exit loop with val as the median

33 **elif** selection == "mean":

34 val = mean((target(e) **for** e **in** training\_data))

35 **elif** selection == "Laplace":

36 val = mean((target(e) **for** e **in** training\_data),**len**(target.frange),1)

37 **elif** selection == "mode":

38 **raise** NotImplementedError("mode")

39 **else**:

40 **raise** RuntimeError("Not valid selection: "+**str**(selection))

41 fun = **lambda** x: val

42 fun. doc = **str**(val)

43 **return** fun

44

45 **def** mean(enum,count=0,**sum**=0):

46 """returns the mean of enumeration enum,

47 count and sum are initial counts and the initial sum.

48 This works for enumerations, even where len() is not defined"""

49 **for** e **in** enum:

50 count += 1

51 **sum** += e

52 **return sum**/count

53

54 **def** target\_counts(target, data\_subset):

55 """returns a value:count dictionary of the count of the number of

56 times target has this value in data\_subset, and the number of examples.

57 """

58 counts = {val:0 **for** val **in** target.frange}

59 total = 0

60 **for** instance **in** data\_subset:

61 total += 1

62 counts[target(instance)] += 1

63 **return** counts, total

### 7.2.1 Testing

#### To test the point prediction, we will first generate some data from a simple (Bernoulli) distribution, where there are two possible values, 0 and 1 for the target feature. Given *prob*, a number in the range [0, 1], this generate some training and test data where *prob* is the probability of each example being 1.

learnNoInputs.py — (continued)

65 **class** Data\_set\_random(Data\_set):

66 """A data set of a {0,1} feature generated randomly given a probability"""

67 **def** init (self, prob, train\_size, test\_size=100):

68 """a data set of with train\_size training examples,

69 test\_size test examples

70 where each examples in generated where prob i the probability of 1

71 """

72 self.max\_display\_level = 0

73 train = [[1] **if** random.random()<prob **else** [0] **for** i **in range**(train\_size)]

74 test = [[1] **if** random.random()<prob **else** [0] **for** i **in range**(test\_size)]

75 Data\_set. init (self, train, test, target\_index=0)

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

learnNoInputs.py — (continued)

77 **def** test\_no\_inputs():

78 num\_samples = 1000 #number of runs to average over

79 test\_size = 100 # number of test examples for each prediction

80 **for** train\_size **in** [1,2,3,4,5,10,20,100,1000]:

81 total\_error = {(select,crit):0

82 **for** select **in** selections

83 **for** crit **in** Data\_set.evaluation\_criteria}

84 **for** sample **in range**(num\_samples): # average over num\_samples

85 p = random.random()

86 data = Data\_set\_random(p, train\_size, test\_size)

87 **for** select **in** selections:

88 prediction = point\_prediction(data.target, data.train, selection=select)

89 **for** ecrit **in** Data\_set.evaluation\_criteria:

90 test\_error = data.evaluate\_dataset(data.test,prediction,ecrit)

91 total\_error[(select,ecrit)] += test\_error

92 **print**("For training size",train\_size,":")

93 **for** ecrit **in** Data\_set.evaluation\_criteria:

94 **print**(" Evaluated according to",ecrit,":")

95 **for** select **in** selections:

96 **print**(" Average error of",select,"is",

97 total\_error[(select,ecrit)]/num\_samples)

98

99 **if** name == " main ":

100 test\_no\_inputs()

## 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, error\_example

12 **from** learnNoInputs **import** point\_prediction, target\_counts, selections

13 **import** math

14

15 **class** DT\_learner(Learner):

16 **def** init (self,

17 dataset,

18 to\_optimize="sum-of-squares",

19 leaf\_selection="mean", # what to use for point prediction at leaves

20 train=None, # used for cross validation

21 min\_number\_examples=10):

22 self.dataset = dataset

23 self.target = dataset.target

24 self.to\_optimize = to\_optimize

25 self.leaf\_selection = leaf\_selection

26 self.min\_number\_examples = min\_number\_examples

27 **if** train **is** None:

28 self.train = self.dataset.train

29 **else**:

30 self.train = train

31

32 **def** learn(self):

33 **return** self.learn\_tree(self.dataset.input\_features, self.train)

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

It doesn’t split if there are no more input features, if there are fewer exam- ples than *min number examples*, if all the examples agree on the value of the target or if the best split makes all examples in the same partition

If it decides to split, it selects the best split and returns the condition to split on (in the variable *split*) and the corresponding partition of the examples.

learnDT.py — (continued)

35 **def** learn\_tree(self, input\_features, data\_subset):

36 """returns a decision tree

37 for input\_features is a set of possible conditions

38 data\_subset is a subset of the data used to build this (sub)tree

39

40 where a decision tree is a function that takes an example and

41 makes a prediction on the target feature

42 """

43 **if** (input\_features **and len**(data\_subset) >= self.min\_number\_examples):

44 first\_target\_val = self.target(data\_subset[0])

45 allagree = **all**(self.target(inst)==first\_target\_val **for** inst **in** data\_subset)

46 **if not** allagree:

47 split, partn = self.select\_split(input\_features, data\_subset)

48 **if** split: # the split succeeded in splitting the data

49 false\_examples, true\_examples = partn

50 rem\_features = [fe **for** fe **in** input\_features **if** fe != split]

51 self.display(2,"Splitting on",split. doc ,"with examples split",

52 **len**(true\_examples),":",**len**(false\_examples))

53 true\_tree = self.learn\_tree(rem\_features,true\_examples)

54 false\_tree = self.learn\_tree(rem\_features,false\_examples)

55 **def** fun(e):

56 **if** split(e):

57 **return** true\_tree(e)

58 **else**:

59 **return** false\_tree(e)

60 #fun = lambda e: true\_tree(e) if split(e) else false\_tree(e)

61 fun. doc = ("if "+split. doc +" then ("+true\_tree. doc +

62 ") else ("+false\_tree. doc +")")

63 **return** fun

64 # don't expand the trees but return a point prediction

65 **return** point\_prediction(self.target, data\_subset, selection=self.leaf\_selection)

learnDT.py — (continued)

67 **def** select\_split(self, input\_features, data\_subset):

68 """finds best feature to split on.

69

70 input\_features is a non-empty list of features.

71 returns feature, partition

72 where feature is an input feature with the smallest error as

73 judged by to\_optimize or

74 feature==None if there are no splits that improve the error

75 partition is a pair (false\_examples, true\_examples) if feature is not None

76 """

77 best\_feat = None # best feature

78 # best\_error = float("inf") # infinity - more than any error

79 best\_error = training\_error(self.dataset, data\_subset, self.to\_optimize)

80 best\_partition = None

81 **for** feat **in** input\_features:

82 false\_examples, true\_examples = partition(data\_subset,feat)

83 **if** false\_examples **and** true\_examples: #both partitons are non-empty

84 err = (training\_error(self.dataset,false\_examples,self.to\_optimize)

85 + training\_error(self.dataset,true\_examples,self.to\_optimize))

86 self.display(3," split on",feat. doc ,"has err=",err,

87 "splits into",**len**(true\_examples),":",**len**(false\_examples))

88 **if** err < best\_error:

89 best\_feat = feat

90 best\_error=err

91 best\_partition = false\_examples, true\_examples

92 self.display(3,"best split is on",best\_feat. doc ,

93 "with err=",best\_error)

94 **return** best\_feat, best\_partition

95

96 **def** partition(data\_subset,feature):

97 """partitions the data\_subset by the feature"""

98 true\_examples = []

99 false\_examples = []

100 **for** example **in** data\_subset:

101 **if** feature(example):

102 true\_examples.append(example)

103 **else**:

104 false\_examples.append(example)

105 **return** false\_examples, true\_examples

106

107

108 **def** training\_error(dataset, data\_subset, to\_optimize):

109 """returns training error for dataset on to\_optimize.

110 This assumes that we choose the best value for the optimization

111 criteria for dataset according to point\_prediction

112 """

113 select\_dict = {"sum-of-squares":"mean", "sum\_absolute":"median",

114 "logloss":"Laplace"} # arbitrary mapping. Perhaps wrong.

115 selection = select\_dict[to\_optimize]

116 predictor = point\_prediction(dataset.target, data\_subset, selection=selection)

117 error = **sum**(error\_example(predictor(example),

118 dataset.target(example),

119 to\_optimize)

120 **for** example **in** data\_subset)

121 **return** error

#### Test cases:

learnDT.py — (continued)

123 **from** learnProblem **import** Data\_set, Data\_from\_file

124

125 **def** test(data):

126 """Prints errors and the trees for various evaluation criteria and ways to select leaves.

127 """

128 **for** crit **in** Data\_set.evaluation\_criteria:

129 **for** leaf **in** selections:

130 tree = DT\_learner(data, to\_optimize=crit, leaf\_selection=leaf).learn()

131 **print**("For",crit,"using",leaf,"at leaves, tree built is:",tree. doc )

132 **if** data.test:

133 **for** ecrit **in** Data\_set.evaluation\_criteria:

134 test\_error = data.evaluate\_dataset(data.test, tree, ecrit)

135 **print**(" Average error for", ecrit,"using",leaf, "at leaves is", test\_error)

136

137 **if** name == " main ":

138 #print("carbool.csv"); test(data = Data\_from\_file('data/carbool.csv', target\_index=-1))

139 # print("SPECT.csv"); test(data = Data\_from\_file('data/SPECT.csv', target\_index=0))

140 **print**("mail\_reading.csv"); test(data = Data\_from\_file('data/mail\_reading.csv', target\_index=-1

141 # print("holiday.csv"); test(data = Data\_from\_file('data/holiday.csv', num\_train=19, target\_in

**Exercise 7.4** 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 7.5** Extend the current algorithm to include in the stopping criterion

* 1. A minimum child size; don’t use a split if one of the children has fewer elements that this.
  2. A depth-bound on the depth of the tree.
  3. 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 7.6** 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 im- provement bound and pseudo-counts make the algorithm work better than with an improvement bound by itself?

**Exercise 7.7** Some people have suggested using information gain (which is equiv- alent to greedy optimization of logloss) as the measure of improvement when building the tree, even in they want to have non-probabilistic predictions in the

final tree. Does this work better than myopically choosing the split that is best for the evaluation criteria we will use to judge the final prediction?

## 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 plot fig 7 15() to produce a graph like Figure 7.15. Note that different runs will produce different graphs, so your graph will not look like the one in the textbook. To try more examples, copy and paste the commented-out commands at the bottom of that file. This requires Python 3 with matplotlib.

The above decision tree overfits the data. One way to determine whether the prediction is overfitting is by cross validation. The code below implements *k*-fold cross validation, which can be used to choose the value of parameters to best fit the training data. If we want to use parameter tuning to improve predictions on a particular data set, we can only use the training data (and not the test data) to tune the parameter.

In *k*-fold cross validation, we partition the training set into *k* approximately equal-sized folds (each fold is an enumeration of examples). For each fold, we train on the other examples, and determine the error of the prediction on that fold. For example, if there are 10 folds, we train on 90% of the data, and then test on remaining 10% of the data. We do this 10 times, so that each example gets used as a test set once, and in the training set 9 times.

The code below creates one copy of the data, and multiple views of the data. For each fold, *fold* enumerates the examples in the fold, and *fold complement* enumerates the examples not in the fold.

learnCrossValidation.py — Cross Validation for Parameter Tuning

11 **from** learnProblem **import** Data\_set, Data\_from\_file, error\_example

12 **from** learnDT **import** DT\_learner

13 **import** matplotlib.pyplot as plt

14 **import** random

15

16 **class** K\_fold\_dataset(**object**):

17 **def** init (self, training\_set, num\_folds):

18 self.data = training\_set.train.copy()

19 self.target = training\_set.target

20 self.input\_features = training\_set.input\_features

21 self.num\_folds = num\_folds

22 random.shuffle(self.data)

23 self.fold\_boundaries = [(**len**(self.data)\*i)//num\_folds

24 **for** i **in range**(0,num\_folds+1)]

25

26 **def** fold(self, fold\_num):

*7.4. Cross Validation and Parameter Tuning* 121

27 **for** i **in range**(self.fold\_boundaries[fold\_num],

28 self.fold\_boundaries[fold\_num+1]):

29 yield self.data[i]

30

31 **def** fold\_complement(self, fold\_num):

32 **for** i **in range**(0,self.fold\_boundaries[fold\_num]):

33 yield self.data[i]

34 **for** i **in range**(self.fold\_boundaries[fold\_num+1],**len**(self.data)):

35 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)

37 **def** validation\_error(self, learner, criterion, \*\*other\_params):

38 error = 0

39 **try**:

40 **for** i **in range**(self.num\_folds):

41 predictor = learner(self, train=**list**(self.fold\_complement(i)),

42 \*\*other\_params).learn()

43 error += **sum**( error\_example(predictor(example),

44 self.target(example),

45 criterion)

46 **for** example **in** self.fold(i))

47 **except** ValueError:

48 **return float**("inf") #infinity

49 **return** error/**len**(self.data)

#### The *plot error* method plots the average error as a function of a the minimun number of examples in decision-tree search, both for the validation set and for the test set. The error on the validation set can be used to tune the parameter

— choose the value of the parameter that minimizes the error. The error on the test set cannot be used to tune the parameters; if is were to be used this way then it cannot be used to test.

learnCrossValidation.py — (continued)

51 **def** plot\_error(data,criterion="sum-of-squares", num\_folds=5, xscale='log'):

52 """Plots the error on the validation set and the test set

53 with respect to settings of the minimum number of examples.

54 xscale should be 'log' or 'linear'

55 """

56 plt.ion()

57 plt.xscale('linear') # change between log and linear scale

58 plt.xlabel("minimum number of examples")

59 plt.ylabel("average "+criterion+" error")

60 folded\_data = K\_fold\_dataset(data, num\_folds)

61 verrors = [] # validation errors

62 terrors = [] # test set errors

63 **for** mne **in range**(1,**len**(data.train)+2):

64 verrors.append(folded\_data.validation\_error(DT\_learner,criterion,

65 min\_number\_examples=mne))

66 tree = DT\_learner(data, criterion, min\_number\_examples=mne).learn()

67 terrors.append(data.evaluate\_dataset(data.test,tree,criterion))

68 plt.plot(**range**(1,**len**(data.train)+2), verrors, ls='-',color='k', label="validation for "+criterion

69 plt.plot(**range**(1,**len**(data.train)+2), terrors, ls='--',color='k', label="test set for "+criterion)

70 plt.legend()

71 plt.draw()

72

73 # Try

74 # data = Data\_from\_file('data/mail\_reading.csv', target\_index=-1)

75 # data = Data\_from\_file('data/SPECT.csv',target\_index=0)

76 # data = Data\_from\_file('data/carbool.csv', target\_index=-1)

77 # plot\_error(data) # warning, may take a long time depending on the dataset

78

79 **def** plot\_fig\_7\_15(): # different runs produce different plots

80 data = Data\_from\_file('data/SPECT.csv',target\_index=0)

81 # data = Data\_from\_file('data/carbool.csv', target\_index=-1)

82 plot\_error(data)

83 # plot\_fig\_7\_15() # warning takes a long time!

## Linear Regression and Classification

#### Here we give a gradient descent searcher for linear regression and classifica- tion.

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):

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.input\_features = dataset.input\_features+[one] # one is defined below

37 self.weights = {feat:random.uniform(-max\_init,max\_init)

38 **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)

40

41 **def** predictor(self,e):

42 """returns the prediction of the learner on example e"""

43 linpred = **sum**(w\*f(e) **for** f,w **in** self.weights.items())

44 **if** self.squashed:

45 **return** sigmoid(linpred)

46 **else**:

47 **return** linpred

48

49 **def** predictor\_string(self, sig\_dig=3):

50 """returns the doc string for the current prediction function

51 sig\_dig is the number of significant digits in the numbers"""

52 doc = "+".join(**str**(**round**(val,sig\_dig))+"\*"+feat. doc

53 **for** feat,val **in** self.weights.items())

54 **if** self.squashed:

55 **return** "sigmoid("+ doc+")"

56 **else**:

57 **return** doc

#### *learn* is the main algorithm of the learner. It does *num iter* steps of gradient descent. The other parameters it gets from the class.

learnLinear.py — (continued)

59 **def** learn(self,num\_iter=100):

60 **for** it **in range**(num\_iter):

61 self.display(2,"prediction=",self.predictor\_string())

62 **for** e **in** self.train:

63 predicted = self.predictor(e)

64 error = self.target(e) - predicted

65 update = self.learning\_rate\*error

66 **for** feat **in** self.weights:

67 self.weights[feat] += update\*feat(e)

68 #self.predictor. doc = self.predictor\_string()

69 #return self.predictor

#### *one* is a function that always returns 1. This is used for one of the input prop- erties.

learnLinear.py — (continued)

71 **def** one(e):

72 "1"

73 **return** 1

*sigmoid*(*x*) is the function

#### 1

1 + *e*−*x*

learnLinear.py — (continued)

75 **def** sigmoid(x):

76 **return** 1/(1+math.exp(-x))

#### The following tests the learner on a data sets. Uncomment the other data sets for different examples.

learnLinear.py — (continued)

78 **from** learnProblem **import** Data\_set, Data\_from\_file

79 **import** matplotlib.pyplot as plt

80 **def** test(\*\*args):

81 data = Data\_from\_file('data/SPECT.csv', target\_index=0)

82 # data = Data\_from\_file('data/mail\_reading.csv', target\_index=-1)

83 # data = Data\_from\_file('data/carbool.csv', target\_index=-1)

84 learner = Linear\_learner(data,\*\*args)

85 learner.learn()

86 **print**("function learned is", learner.predictor\_string())

87 **for** ecrit **in** Data\_set.evaluation\_criteria:

88 test\_error = data.evaluate\_dataset(data.test, learner.predictor, ecrit)

89 **print**(" Average", ecrit, "error is", test\_error)

#### The following plots the errors on the training and test sets as a function of the number of steps of gradient descent.

learnLinear.py — (continued)

91 **def** plot\_steps(learner=None,

92 data = None,

93 criterion="sum-of-squares",

94 step=1,

95 num\_steps=1000,

96 log\_scale=True,

97 label=""):

98 """

99 plots the training and test error for a learner.

100 data is the

101 learner\_class is the class of the learning algorithm

102 criterion gives the evaluation criterion plotted on the y-axis

103 step specifies how many steps are run for each point on the plot

104 num\_steps is the number of points to plot

105

106 """

107 plt.ion()

108 plt.xlabel("step")

109 plt.ylabel("Average "+criterion+" error")

110 **if** log\_scale:

111 plt.xscale('log') #plt.semilogx() #Makes a log scale

112 **else**:

113 plt.xscale('linear')

114 **if** data **is** None:

115 data = Data\_from\_file('data/holiday.csv', num\_train=19, target\_index=-1)

116 #data = Data\_from\_file('data/SPECT.csv', target\_index=0)

117 # data = Data\_from\_file('data/mail\_reading.csv', target\_index=-1)

118 # data = Data\_from\_file('data/carbool.csv', target\_index=-1)

119 random.seed(None) # reset seed

120 **if** learner **is** None:

121 learner = Linear\_learner(data)

122 train\_errors = []

123 test\_errors = []

124 **for** i **in range**(1,num\_steps+1,step):

125 test\_errors.append(data.evaluate\_dataset(data.test, learner.predictor, criterion))

126 train\_errors.append(data.evaluate\_dataset(data.train, learner.predictor, criterion))

127 learner.display(2, "Train error:",train\_errors[-1],

128 "Test error:",test\_errors[-1])

129 learner.learn(num\_iter=step)

130 plt.plot(**range**(1,num\_steps+1,step),train\_errors,ls='-',c='k',label="training errors")

131 plt.plot(**range**(1,num\_steps+1,step),test\_errors,ls='--',c='k',label="test errors")

132 plt.legend()

133 plt.draw()

134 learner.display(1, "Train error:",train\_errors[-1],

135 "Test error:",test\_errors[-1])

136

137 **if** name == " main ":

138 test()

139

140 # This generates the figure

141 # from learnProblem import Data\_set\_augmented,prod\_feat

142 # data = Data\_from\_file('data/SPECT.csv', prob\_test=0.5, target\_index=0)

143 # dataplus = Data\_set\_augmented(data,[],[prod\_feat])

144 # plot\_steps(data=data,num\_steps=10000)

145 # plot\_steps(data=dataplus,num\_steps=10000) # warning very slow

**Exercise 7.8** The squashed learner only makes predictions in the range (0, 1). If the output values are 1, 2, 3, 4 there is no use prediction less than 1 or greater than 4. Change the squashed learner so that it can learn values in the range (1, 4). Test it on the file 'data/car.csv'.

{ }

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

learnLinear.py — (continued)

146 **def** arange(start,stop,step):

147 """returns enumeration of values in the range [start,stop) separated by step.

148 like the built-in range(start,stop,step) but allows for integers and floats.

149 Note that rounding errors are expected with real numbers.

150 """

151 **while** start<stop:

152 yield start

153 start += step

154

155 **def** plot\_prediction(learner=None,

156 data = None,

157 minx = 0,

158 maxx = 5,

159 step\_size = 0.01, # for plotting

160 label="function"):

161 plt.ion()

162 plt.xlabel("x")

163 plt.ylabel("y")

164 **if** data **is** None:

165 data = Data\_from\_file('data/simp\_regr.csv', prob\_test=0,

166 boolean\_features=False, target\_index=-1)

167 **if** learner **is** None:

168 learner = Linear\_learner(data,squashed=False)

169 learner.learning\_rate=0.001

170 learner.learn(100)

171 learner.learning\_rate=0.0001

172 learner.learn(1000)

173 learner.learning\_rate=0.00001

174 learner.learn(10000)

175 learner.display(1,"function learned is", learner.predictor\_string(),

176 "error=",data.evaluate\_dataset(data.train, learner.predictor, "sum-of-squares"))

177 plt.plot([e[0] **for** e **in** data.train],[e[-1] **for** e **in** data.train],"bo",label="data")

178 plt.plot(**list**(arange(minx,maxx,step\_size)),[learner.predictor([x])

179 **for** x **in** arange(minx,maxx,step\_size)],

180 label=label)

181 plt.legend()

182 plt.draw()

|  |  |  |
| --- | --- | --- |
| 184  185  186 |  | learnLinear.py — (continued)  **from** learnProblem **import** Data\_set\_augmented, power\_feat  **def** plot\_polynomials(data=None,  learner\_class = Linear\_learner, |
| 187 |  | max\_degree=5, |
| 188 |  | minx = 0, |
| 189 |  | maxx = 5, |
| 190 |  | num\_iter = 100000, |
| 191 |  | learning\_rate = 0.0001, |
| 192 |  | step\_size = 0.01, # for plotting |
| 193 |  | ): |
| 194 |  | plt.ion() |
| 195 |  | plt.xlabel("x") |
| 196 |  | plt.ylabel("y") |
| 197 |  | **if** data **is** None: |
| 198 |  | data = Data\_from\_file('data/simp\_regr.csv', prob\_test=0, |
| 199 |  | boolean\_features=False, target\_index=-1) |
| 200 |  | plt.plot([e[0] **for** e **in** data.train],[e[-1] **for** e **in** data.train],"ko",label="data") |

201 x\_values = **list**(arange(minx,maxx,step\_size))

202 line\_styles = ['-','--','-.',':']

203 colors = ['0.5','k','k','k','k']

204 **for** degree **in range**(max\_degree):

205 data\_aug = Data\_set\_augmented(data,[power\_feat(n) **for** n **in range**(1,degree+1)],

206 include\_orig=False)

207 learner = learner\_class(data\_aug,squashed=False)

208 learner.learning\_rate=learning\_rate

209 learner.learn(num\_iter)

210 learner.display(1,"For degree",degree,

211 "function learned is", learner.predictor\_string(),

212 "error=",data.evaluate\_dataset(data.train, learner.predictor, "sum-of-squares")

|  |  |
| --- | --- |
| 213 | ls = line\_styles[degree % **len**(line\_styles)] |
| 214 | col = colors[degree % **len**(colors)] |
| 215 | plt.plot(x\_values,[learner.predictor([x]) **for** x **in** x\_values], linestyle=ls, color=col, |
| 216 | label="degree="+**str**(degree)) |
| 217 | plt.legend(loc='upper left') |
| 218 | plt.draw() |
| 219 |  |
| 220 | # Try: |
| 221 | # plot\_prediction() |
| 222 | # plot\_polynomials() |
| 223 | #data = Data\_from\_file('data/mail\_reading.csv', target\_index=-1) |
| 224 | #plot\_prediction(data=data) |

7.5.1 Batched Stochastic Gradient Descent

#### This implements batched stochastic gradient descent. If the batch size is 1, it can be simplified by not storing the differences in *d*, but applying them directly; this would the be equivalent to the original code!

This overrides the learner *Linear learner*. Note that the comparison with regular gradient descent is unfair as the number of updates per step is not the same. (How could it me made more fair?)

learnLinearBSGD.py — Linear Learner with Batched Stochastic Gradient Descent

11 **from** learnLinear **import** Linear\_learner

12 **import** random, math

13

14 **class** Linear\_learner\_bsgd(Linear\_learner):

15 **def** init (self, \*args, batch\_size=10, \*\*kargs):

16 Linear\_learner. init (self, \*args, \*\*kargs)

17 self.batch\_size = batch\_size

18

19 **def** learn(self,num\_iter=None):

20 **if** num\_iter **is** None:

21 num\_iter = self.number\_iterations

22 batch\_size = **min**(self.batch\_size, **len**(self.train))

23 d = {feat:0 **for** feat **in** self.weights}

24 **for** it **in range**(num\_iter):

25 self.display(2,"prediction=",self.predictor\_string())

26 **for** e **in** random.sample(self.train, batch\_size):

27 predicted = self.predictor(e)

28 error = self.target(e) - predicted

29 update = self.learning\_rate\*error

30 **for** feat **in** self.weights:

31 d[feat] += update\*feat(e)

32 **for** feat **in** self.weights:

33 self.weights[feat] += d[feat]

34 d[feat]=0

35

36 # from learnLinear import plot\_steps

37 # from learnProblem import Data\_from\_file

38 # data = Data\_from\_file('data/holiday.csv', target\_index=-1)

39 # learner = Linear\_learner\_bsgd(data)

40 # plot\_steps(learner = learner, data=data)

41

42 # to plot polynomials with batching (compare to SGD)

43 # from learnLinear import plot\_polynomials

44 # plot\_polynomials(learner\_class = Linear\_learner\_bsgd)

## Deep Neural Network Learning

#### This provides a modular implementation that implements the layers modu- larly. Layers can easily be configured in many configurations. A layer needs to implement a function to compute the output values from the inputs and a way to back-propagate the error.

learnNN.py — Neural Network Learning

11 **from** learnProblem **import** Learner, Data\_set, Data\_from\_file

12 **from** learnLinear **import** sigmoid, one

13 **import** random, math

14

15 **class** Layer(**object**):

16 **def** init (self,nn,num\_outputs=None):

17 """Given a list of inputs, outputs will produce a list of length num\_outputs.

18 nn is the neural network this is part of

19 num outputs is the number of outputs for this layer.

20 """

21 self.nn = nn

22 self.num\_inputs = nn.num\_outputs # output of nn is the input to this layer

23 **if** num\_outputs:

24 self.num\_outputs = num\_outputs

25 **else**:

26 self.num\_outputs = nn.num\_outputs # same as the inputs

27

28 **def** output\_values(self,input\_values):

29 """Return the outputs for this layer for the given input values.

30 input\_values is a list of the inputs to this layer (of length num\_inputs)

31 returns a list of length self.num\_outputs

32 """

33 **raise** NotImplementedError("output\_values") # abstract method

34

35 **def** backprop(self,errors):

36 """Backpropagate the errors on the outputs, return the errors on the inputs.

37 errors is a list of errors for the outputs (of length self.num\_outputs).

38 Return the errors for the inputs to this layer (of length self.num\_inputs).

39 You can assume that this is only called after corresponding output\_values,

40 and it can remember information information required for the backpropagation.

41 """

42 **raise** NotImplementedError("backprop") # abstract method

#### A linear layer maintains an array of weights. *self* .*weights*[*o*][*i*] is the weight between input *i* and output *o*. A 1 is added to the inputs.

learnNN.py — (continued)

44 **class** Linear\_complete\_layer(Layer):

45 """a completely connected layer"""

46 **def** init (self, nn, num\_outputs, max\_init=0.2):

47 """A completely connected linear layer.

48 nn is a neural network that the inputs come from

49 num\_outputs is the number of outputs

50 max\_init is the maximum value for random initialization of parameters

51 """

52 Layer. init (self, nn, num\_outputs)

53 # self.weights[o][i] is the weight between input i and output o

54 self.weights = [[random.uniform(-max\_init, max\_init)

55 **for** inf **in range**(self.num\_inputs+1)]

56 **for** outf **in range**(self.num\_outputs)]

57

58 **def** output\_values(self,input\_values):

59 """Returns the outputs for the input values.

60 It remembers the values for the backprop.

61

62 Note in self.weights there is a weight list for every output,

63 so wts in self.weights effectively loops over the outputs.

64 """

65 self.inputs = input\_values + [1]

66 **return** [**sum**(w\*val **for** (w,val) **in zip**(wts,self.inputs))

67 **for** wts **in** self.weights]

68

69 **def** backprop(self,errors):

70 """Backpropagate the errors, updating the weights and returning the error in its inputs.

71 """

72 input\_errors = [0]\*(self.num\_inputs+1)

73 **for** out **in range**(self.num\_outputs):

74 **for** inp **in range**(self.num\_inputs+1):

75 input\_errors[inp] += self.weights[out][inp] \* errors[out]

76 self.weights[out][inp] += self.nn.learning\_rate \* self.inputs[inp] \* errors[out]

77 **return** input\_errors[:-1] # remove the error for the "1"

learnNN.py — (continued)

79 **class** Sigmoid\_layer(Layer):

80 """sigmoids of the inputs.

81 The number of outputs is equal to the number of inputs.

82 Each output is the sigmoid of its corresponding input.

83 """

84 **def** init (self, nn):

85 Layer. init (self, nn)

86

87 **def** output\_values(self,input\_values):

88 """Returns the outputs for the input values.

89 It remembers the output values for the backprop.

90 """

91 self.outputs= [sigmoid(inp) **for** inp **in** input\_values]

92 **return** self.outputs

93

94 **def** backprop(self,errors):

95 """Returns the derivative of the errors"""

96 **return** [e\*out\*(1-out) **for** e,out **in zip**(errors, self.outputs)]

|  |  |  |
| --- | --- | --- |
| 98  99  100  101  102  103  104  105  106  107  108  109  110  111 |  | learnNN.py — (continued)  **class** ReLU\_layer(Layer):  """Rectified linear unit (ReLU) f(z) = max(0, z).  The number of outputs is equal to the number of inputs. """  **def** init (self, nn): Layer. init (self, nn)  **def** output\_values(self,input\_values): """Returns the outputs for the input values.  It remembers the input values for the backprop. """  self.input\_values = input\_values  self.outputs= [**max**(0,inp) **for** inp **in** input\_values]  **return** self.outputs |
| 112 |  |  |
| 113 |  | **def** backprop(self,errors): |
| 114 |  | """Returns the derivative of the errors""" |
| 115 |  | **return** [e **if** inp>0 **else** 0 **for** e,inp **in zip**(errors, self.input\_values)] |

learnNN.py — (continued)

117 **class** NN(Learner):

118 **def** init (self, dataset, learning\_rate=0.1):

119 self.dataset = dataset

120 self.learning\_rate = learning\_rate

121 self.input\_features = dataset.input\_features

122 self.num\_outputs = **len**(self.input\_features)

123 self.layers = []

124

125 **def** add\_layer(self,layer):

|  |  |  |
| --- | --- | --- |
| 126 |  | """add a layer to the network. |
| 127 |  | Each layer gets values from the previous layer. |
| 128 |  | """ |
| 129 |  | self.layers.append(layer) |
| 130 |  | self.num\_outputs = layer.num\_outputs |
| 131 |  |  |
| 132 | **def** | predictor(self,ex): |
| 133 |  | """Predicts the value of the first output feature for example ex. |
| 134 |  | """ |
| 135 |  | values = [f(ex) **for** f **in** self.input\_features] |
| 136  137 |  | **for** layer **in** self.layers:  values = layer.output\_values(values) |
| 138 |  | **return** values[0] |
| 139 |  |  |
| 140 | **def** | predictor\_string(self): |
| 141 |  | **return** "not implemented" |

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

learnNN.py — (continued)

143

144 **def** learn(self,num\_iter):

145 """Learns parameters for a neural network using stochastic gradient decent.

146 num\_iter is the number of iterations

147 """

148 **for** i **in range**(num\_iter):

149 **for** e **in** random.sample(self.dataset.train,**len**(self.dataset.train)):

150 # compute all outputs

151 values = [f(e) **for** f **in** self.input\_features]

152 **for** layer **in** self.layers:

153 values = layer.output\_values(values)

154 # backpropagate

155 errors = self.sum\_squares\_error([self.dataset.target(e)],values)

156 **for** layer **in reversed**(self.layers):

157 errors = layer.backprop(errors)

158

159 **def** sum\_squares\_error(self,observed,predicted):

160 """Returns the errors for each of the target features.

161 """

162 **return** [obsd-pred **for** obsd,pred **in zip**(observed,predicted)]

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

learnNN.py — (continued)

165 data = Data\_from\_file('data/mail\_reading.csv', target\_index=-1)

166 #data = Data\_from\_file('data/mail\_reading\_consis.csv', target\_index=-1)

167 #data = Data\_from\_file('data/SPECT.csv', prob\_test=0.5, target\_index=0)

168 #data = Data\_from\_file('data/holiday.csv', target\_index=-1) #, num\_train=19)

169 nn1 = NN(data)

170 nn1.add\_layer(Linear\_complete\_layer(nn1,3))

171 nn1.add\_layer(Sigmoid\_layer(nn1)) # comment this or the next

172 # nn1.add\_layer(ReLU\_layer(nn1))

173 nn1.add\_layer(Linear\_complete\_layer(nn1,1))

174 nn1.add\_layer(Sigmoid\_layer(nn1))

175 nn1.learning\_rate=0.1

176 #nn1.learn(100)

177

178 **from** learnLinear **import** plot\_steps

179 **import** time

180 start\_time = time.perf\_counter()

181 plot\_steps(learner = nn1, data = data, num\_steps=10000)

182 **for** eg **in** data.train:

183 **print**(eg,nn1.predictor(eg))

184 end\_time = time.perf\_counter()

185 **print**("Time:", end\_time - start\_time)

**Exercise 7.9** In the definition of *nn*1 above, for each of the following, first hy- pothesize 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.

1. Which fits the data better, having a sigmoid layer or a ReLU layer after the first linear layer?
2. Which is faster, having a sigmoid layer or a ReLU layer after the first linear layer?
3. 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?
4. What happens if you have neither the sigmoid layer nor a ReLU layer after the first linear layer?
5. What happens if you have a ReLU layer then a sigmoid layer after the first linear layer and before the second linear layer?

**Exercise 7.10** Do some

#### It is even possible to define a perceptron layer. Warning: you may need to change the learning rate to make this work. Should I add it into the code? It doesn’t follow the official line.

class PerceptronLayer(Layer): def init (self, nn):

Layer. init (self, nn)

def output\_values(self,input\_values):

"""Returns the outputs for the input values. """

self.outputs= [1 if inp>0 else -1 for inp in input\_values] return self.outputs

def backprop(self,errors): """Pass the errors through""" return errors

## Boosting

The following code implements functional gradient boosting for regression.

A Boosted dataset is created from a base dataset by subtracting the pre- diction 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, in- dependent on the size of the data set.

learnBoosting.py — Functional Gradient Boosting

11 **from** learnProblem **import** Data\_set, Learner

12

13 **class** Boosted\_dataset(Data\_set):

14 **def** init (self, base\_dataset, offset\_fun):

15 """new dataset which is like base\_dataset,

16 but offset\_fun(e) is subtracted from the target of each example e

17 """

18 self.base\_dataset = base\_dataset

19 self.offset\_fun = offset\_fun

20 Data\_set. init (self, base\_dataset.train, base\_dataset.test,

21 base\_dataset.prob\_test, base\_dataset.target\_index)

22

23 **def** create\_features(self):

24 self.input\_features = self.base\_dataset.input\_features

25 **def** newout(e):

26 **return** self.base\_dataset.target(e) - self.offset\_fun(e)

27 newout.frange = self.base\_dataset.target.frange

28 self.target = newout

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

learnBoosting.py — (continued)

30 **class** Boosting\_learner(Learner):

31 **def** init (self, dataset, base\_learner\_class):

32 self.dataset = dataset

33 self.base\_learner\_class = base\_learner\_class

34 mean = **sum**(self.dataset.target(e)

35 **for** e **in** self.dataset.train)/**len**(self.dataset.train)

36 self.predictor = **lambda** e:mean # function that returns mean for each example

37 self.predictor. doc = "lambda e:"+**str**(mean)

38 self.offsets = [self.predictor]

39 self.errors = [data.evaluate\_dataset(data.test, self.predictor, "sum-of-squares")]

40 self.display(1,"Predict mean test set error=", self.errors[0] )

41

42

43 **def** learn(self, num\_ensemble=10):

44 """adds num\_ensemble learners to the ensemble.

45 returns a new predictor.

46 """

47 **for** i **in range**(num\_ensemble):

48 train\_subset = Boosted\_dataset(self.dataset, self.predictor)

49 learner = self.base\_learner\_class(train\_subset)

50 new\_offset = learner.learn()

51 self.offsets.append(new\_offset)

52 **def** new\_pred(e, old\_pred=self.predictor, off=new\_offset):

53 **return** old\_pred(e)+off(e)

54 self.predictor = new\_pred

55 self.errors.append(data.evaluate\_dataset(data.test, self.predictor,"sum-of-squares"))

56 self.display(1,"After Iteration",**len**(self.offsets)-1,"test set error=", self.errors[-1])

57 **return** self.predictor

#### For testing, *sp DT learner* returns a function that constructs a decision tree learner where the minimum number of examples is a proportion of the number of training examples. The value of 0.9 tends to have one split, and a value of 0.5 tends to have two splits (but test it). Thus this can be used to construct small decision trees that can be used as weak learners.

learnBoosting.py — (continued)

59 # Testing

60

61 **from** learnDT **import** DT\_learner

62 **from** learnProblem **import** Data\_set, Data\_from\_file

63

64 **def** sp\_DT\_learner(min\_prop=0.9):

65 **def** make\_learner(dataset):

66 mne = **len**(dataset.train)\*min\_prop

67 **return** DT\_learner(dataset,min\_number\_examples=mne)

68 **return** make\_learner

69

70 data = Data\_from\_file('data/carbool.csv', target\_index=-1)

71 #data = Data\_from\_file('data/SPECT.csv', target\_index=0)

72 #data = Data\_from\_file('data/mail\_reading.csv', target\_index=-1)

73 #data = Data\_from\_file('data/holiday.csv', num\_train=19, target\_index=-1)

74 learner9 = Boosting\_learner(data, sp\_DT\_learner(0.9))

75 #learner7 = Boosting\_learner(data, sp\_DT\_learner(0.7))

76 #learner5 = Boosting\_learner(data, sp\_DT\_learner(0.5))

77 predictor9 =learner9.learn(10)

78 **for** i **in** learner9.offsets: **print**(i. doc )

79 **import** matplotlib.pyplot as plt

80

81 **def** plot\_boosting(data,steps=10, thresholds=[0.5,0.1,0.01,0.001], markers=['-','--','-.',':'] ):

82 learners = [Boosting\_learner(data, sp\_DT\_learner(th)) **for** th **in** thresholds]

83 predictors = [learner.learn(steps) **for** learner **in** learners]

84 plt.ion()

85 plt.xscale('linear') # change between log and linear scale

86 plt.xlabel("number of trees")

87 plt.ylabel(" error")

88 **for** (learner,(threshold,marker)) **in zip**(learners,**zip**(thresholds,markers)):

89 plt.plot(**range**(**len**(learner.errors)), learner.errors, ls=marker,c='k',

90 label=**str**(**round**(threshold\*100))+"% min example threshold")

91 plt.legend()

92 plt.draw()

93

94 # plot\_boosting(data)

# Chapter 8

Reasoning Under Uncertainty

## 8.1 Representing Probabilistic Models

#### In the implementation of probabilistic models we will assume that the variables are objects, rather than the strings we used for CSPs. (Note that in the CSP code variables could be anything; we just used strings for the examples.) We use a class here because it is more amenable to extend to richer models, such as when we introduce time.

A variable consists of a name and a domain. The domain of a variable is a list or a tuple, as the ordering will matter in the representation of factors. The code below internally uses the index of each value. We define a function *val to index* that maps from the value to the index.

probVariables.py — Probabilistic Variables

11 **class** Variable(**object**):

12 """A random variable.

13 name (string) - name of the variable

14 domain (list) - a list of the values for the variable.

15 Variables are ordered according to their name.

16 """

17

18 **def** init (self,name,domain):

19 self.name = name

20 self.size = **len**(domain)

21 self.domain = domain

22 self.val\_to\_index = {} # map from domain to index

23 **for** i,val **in enumerate**(domain):

24 self.val\_to\_index[val]=i

25

26 **def** str (self):

27 **return** self.name

#### 137

|  |  |  |  |
| --- | --- | --- | --- |
| *A* | *B* | *C* | Value |
| 0 | a | s | *v*0 |
| 0 | a | t | *v*1 |
| 0 | b | s | *v*2 |
| 0 | b | t | *v*3 |
| 0 | c | s | *v*4 |
| 0 | c | t | *v*5 |
| 1 | a | s | *v*6 |
| 1 | a | t | *v*7 |
| 1 | b | s | *v*8 |
| 1 | b | t | *v*9 |
| 1 | c | s | *v*10 |
| 1 | c | t | *v*11 |

Figure 8.1: A representation for a factor for the variable ordering *A*, *B*, *C*

28

29 **def** repr (self):

30 **return** "Variable('"+self.name+"')"

## 8.2 Factors

#### Factors are functions from variables into values. The main problem with vari- able elimination is the amount of space used, because it saves the intermedi- ate factors. (If instead it recomputed factors rather than saving the factors, it would be effectively enumerating the worlds, and so would be exponential in the number of variables). We only want to store the list of numbers, with as little bookkeeping as possible.

A total ordering of the variables, and a total ordering of the values in the domains of the variables induces a total ordering of the values of the factor according to the lexicographic ordering. E.g., suppose the domain of *A* is [0, 1],

domain of *B* is [j*a*j,j *b*j,j *c*j], and the domain of *C* is [j*s*j,j *t*j], the ordering [*A*, *B*, *C*]

#### of variables induces an ordering on the values of the factor, as in Figure [8.1.](#_bookmark137) We just need to store the list of variables and the *vi*s. For any assignment to *A*, *B* and *C*, we can compute the index of the value for that assignment. *A* = *a*, *B* =

*b*, *C* = *c* is stored at location *a*j 6 + *b*j 2 + *c*j, where *a*j is *A*.*val to index*[*a*], and

∗ ∗

similarly for *b*j and *c*j.

probFactors.py — Factor manipulation for graphical models

11 **from** functools **import reduce**

12 #from probVariables import Variable

13

14 **class** Factor(**object**):

15 nextid=0 # each factor has a unique identifier; for printing

16

17 **def** init (self,variables):

18 """variables is the ordered list of variables

19 """

20 self.variables = variables # ordered list of variables

21 # Compute the size and the offsets for the variables

22 self.var\_offsets = {}

23 self.size = 1

24 **for** i **in range**(**len**(variables)-1,-1,-1):

25 self.var\_offsets[variables[i]]=self.size

26 self.size \*= variables[i].size

27 self.**id** = Factor.nextid

28 self.name = "f"+**str**(self.**id**)

29 Factor.nextid += 1

#### For each factor, *get value* returns the value of the factor for an assignment. An **assignment** is a variable:value dictionary. The assignment must include all of the variables involved in the factor, and can include variables not in the factor. This needs to be defined for every subclass.

probFactors.py — (continued)

31 **def** get\_value(self,assignment):

32 **raise** NotImplementedError("get\_value") # abstract method

#### The methods *str* and *brief* return string representations of the factor, as a table or just as a name with the variables it is a factor on.

probFactors.py — (continued)

34 **def** str (self, variables=None):

35 """returns a string representation of the factor.

36 Allows for an arbitrary variable ordering.

37 variables is a list of the variables in the factor

38 (can contain other variables)"""

39 **if** variables==None:

40 variables = self.variables

41 **else**:

42 variables = [v **for** v **in** variables **if** v **in** self.variables]

|  |  |  |
| --- | --- | --- |
| 43 |  | res = "" |
| 44 |  | **for** v **in** variables: |
| 45 |  | res += **str**(v) + "\t" |
| 46 |  | res += self.name+"\n" |
| 47 |  | **for** i **in range**(self.size): |
| 48 |  | asst = self.index\_to\_assignment(i) |
| 49 |  | **for** v **in** variables: |
| 50 |  | res += **str**(asst[v])+"\t" |
| 51 |  | res += **str**(self.get\_value(asst)) |
| 52 |  | res += "\n" |
| 53 |  | **return** res |
| 54 |  |  |
| 55 | **def** | brief(self): |
| 56 |  | """returns a string representing a summary of the factor""" |

57 res = self.name+"("

58 **for** i **in range**(0,**len**(self.variables)-1):

59 res += **str**(self.variables[i])+","

60 **if len**(self.variables)>0:

61 res += **str**(self.variables[**len**(self.variables)-1])

62 res += ")"

63 **return** res

The methods *assignment to index* and *index to assignment* map between the as- signments of values to variables and the index of where that assignment would be stored.

probFactors.py — (continued)

65 **def** assignment\_to\_index(self,assignment):

66 """returns the index where the variable:value assignment is stored"""

67 index = 0

68 **for** var **in** self.variables:

69 index += var.val\_to\_index[assignment[var]]\*self.var\_offsets[var]

70 **return** index

71

72 **def** index\_to\_assignment(self,index):

73 """gives a dict representation of the variable assignment for index

74 """

75 asst = {}

76 **for** i **in range**(**len**(self.variables)-1,-1,-1):

77 asst[self.variables[i]] = self.variables[i].domain[index % self.variables[i].size]

78 index = index // self.variables[i].size

79 **return** asst

A *Factor stored* is a factor that has the values stored in a list.

probFactors.py — (continued)

81 **class** Factor\_stored(Factor):

82 **def** init (self,variables,values):

83 Factor. init (self, variables)

84 self.values = values

85

86 **def** get\_value(self,assignment):

87 **return** self.values[self.assignment\_to\_index(assignment)]

#### A *Factor observed* is a factor that is the result of some observations on an- other 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)

89 **class** Factor\_observed(Factor):

90 **def** init (self,factor,obs):

91 Factor. init (self, [v **for** v **in** factor.variables **if** v **not in** obs])

92 self.observed = obs

93 self.orig\_factor = factor

94

95 **def** get\_value(self,assignment):

96 ass = assignment.copy()

97 **for** ob **in** self.observed:

98 ass[ob]=self.observed[ob]

99 **return** self.orig\_factor.get\_value(ass)

#### A *Factor sum* is a factor that is the result of summing out a variable from the product of other factors. Ie., it constructs a representation of:

∑ ∏ *f* .

*var f* ∈*factors*

#### 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.

|  |  |  |  |
| --- | --- | --- | --- |
| 101  102  103 |  | **class** F  **def** | probFactors.py — (continued)  actor\_sum(Factor\_stored):  init (self,var,factors): self.var\_summed\_out = var |
| 104 |  |  | self.factors = factors |
| 105 |  |  | **vars** = [] |
| 106 |  |  | **for** fac **in** factors: |
| 107 |  |  | **for** v **in** fac.variables: |
| 108 |  |  | **if** v **is not** var **and** v **not in vars**: |
| 109 |  |  | **vars**.append(v) |
| 110 |  |  | Factor\_stored. init (self,**vars**,None) |
| 111 |  |  | self.values = [None]\*self.size |
| 112 |  |  |  |
| 113 |  | **def** | get\_value(self,assignment): |
| 114 |  |  | """lazy implementation: if not saved, compute it. Return saved value""" |
| 115 |  |  | index = self.assignment\_to\_index(assignment) |
| 116 |  |  | **if** self.values[index]: |
| 117  118  119  120  121  122  123  124  125  126  127  128 | **return** self.values[index]  **else**:  total = 0  new\_asst = assignment.copy()  **for** val **in** self.var\_summed\_out.domain: new\_asst[self.var\_summed\_out] = val prod = 1  **for** fac **in** self.factors:  prod \*= fac.get\_value(new\_asst) total += prod  self.values[index] = total  **return** total | | |

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

probFactors.py — (continued)

130 **def** factor\_times(variable,factors):

131 """when factors are factors just on variable (or on no variables)"""

132 prods= []

133 facs = [f **for** f **in** factors **if** variable **in** f.variables]

134 **for** val **in** variable.domain:

135 prod = 1

136 ast = {variable:val}

137 **for** f **in** facs:

138 prod \*= f.get\_value(ast)

139 prods.append(prod)

140 **return** prods

#### *Prob* is a factor that represents a conditional probability.

probFactors.py — (continued)

142 **class** Prob(Factor\_stored):

143 """A factor defined by a conditional probability table"""

144 **def** init (self,var,pars,cpt):

145 """Creates a factor from a conditional probability table, cptf.

146 The cpt values are assumed to be for the ordering par+[var]

147 """

148 Factor\_stored. init (self,pars+[var],cpt)

149 self.child = var

150 self.parents = pars

151 **assert** self.size==**len**(cpt),"Table size incorrect "+**str**(self)

#### *cond dist* returns the probability distribution of the child given values from the parent. This code is based on *assignment to index*. Similarly, *cont prob* returns the probability that the child has a particular value given an assignment of values to the parents. In both of these *par assignment* is a dict that assigns all of the parents (and can also assign other variables, but these are ignored).

probFactors.py — (continued)

153 **def** cond\_dist(self,par\_assignment):

154 """returns the distribution (a val:prob dictionary) over the child given

155 assignment to the parents

156

157 par\_assignment is a variable:value dictionary that assigns values to parents

158 """

159 index = 0

160 **for** var **in** self.parents:

161 index += var.val\_to\_index[par\_assignment[var]]\*self.var\_offsets[var]

162 # index is the position where the disgribution starts

163 **return** {self.child.domain[i]:self.values[index+i] **for** i **in range**(**len**(self.child.domain))}

164

165 **def** cond\_prob(self,par\_assignment,child\_value):

166 """returns the probability child has child\_value given

167 assignment to the parents

168

169 par\_assignment is a variable:value dictionary that assigns values to parents

##### Graphical Models 143

170 child\_value is a value to the child

171 """

172 index = self.child.val\_to\_index[child\_value]

173 **for** var **in** self.parents:

174 index += var.val\_to\_index[par\_assignment[var]]\*self.var\_offsets[var]

175 **return** self.values[index]

#### A *Factor rename* is a factor that is the result renaming the variables in the factor. It takes a factor, *fac*, and a *new* : *old* dictionary, where *new* is the name of a variable in the resulting factor and *old* is the corresponding name in *fac*. This assumes that the all variables are renamed.

probFactors.py — (continued)

177 **class** Factor\_rename(Factor):

178 **def** init (self,fac,renaming):

179 Factor. init (self,**list**(renaming.keys()))

180 self.orig\_fac = fac

181 self.renaming = renaming

182

183 **def** get\_value(self,assignment):

184 **return** self.orig\_fac.get\_value({self.renaming[var]:val

185 **for** (var,val) **in** assignment.items()

186 **if** var **in** self.variables})

## 8.3 Graphical Models

#### A graphical model consists of a set of variables and a set of factors. A be- lief network is a graphical model where all of the factors represent conditional probabilities. There are some operations (such as pruning variables) which are applicable to belief networks, but are not applicable to more general models. At the moment, we will treat them as the same.

probGraphicalModels.py — Graphical Models and Belief Networks

11 **class** Graphical\_model(**object**):

12 """The class of graphical models.

13 A graphical model consists of a set of variables and a set of factors.

14

15 List vars is a list of variables

16 List factors is a list of factors

17 """

18 **def** init (self,**vars**=None,factors=None):

19 self.variables = **vars**

20 self.factors = factors

#### A belief network is a graphical model where all of the factors are condi- tional probabilities, and every variable has a conditional probability. This only checks the first condition:

probGraphicalModels.py — (continued)

22 **class** Belief\_network(Graphical\_model):

23 """The class of belief networks."""

24

25 **def** init (self,**vars**=None,factors=None):

26 """vars is a list of variables

27 factors is a list of factors. Here we assume that all of the factors are instances of Prob.

28 """

29 Graphical\_model. init (self,**vars**,factors)

30 **assert all**(**isinstance**(f,Prob) **for** f **in** factors) **if** factors **else** True

#### Each of the inference methods implements the query method that com- putes the posterior probability of a variable given a dictionary of variable:value observations. These are all Displayable because they implement the *display* method which is currently text-based.

probGraphicalModels.py — (continued)

32 **from** display **import** Displayable

33

34 **class** Inference\_method(Displayable):

35 """The abstract class of graphical model inference methods"""

36 **def** query(self,qvar,obs={}):

37 **raise** NotImplementedError("Inference\_method query") # abstract method

#### The first example belief network is a simple chain *A* −→ *B* −→ *C*.

probGraphicalModels.py — (continued)

39 **from** probVariables **import** Variable

40 **from** probFactors **import** Prob

41

42 boolean = [False, True]

43 A = Variable("A", boolean)

44 B = Variable("B", boolean)

45 C = Variable("C", boolean)

46

47 f\_a = Prob(A,[],[0.4,0.6])

48 f\_b = Prob(B,[A],[0.9,0.1,0.2,0.8])

49 f\_c = Prob(C,[B],[0.5,0.5,0.3,0.7])

50

51 bn1 = Belief\_network([A,B,C],[f\_a,f\_b,f\_c])

The second Bayesian network is the report-of-leaving example from Poole and Mackworth, Artificial Intelligence, 2010 [http://artint.info](http://artint.info/). This is Example

* 1. (page 236) shown in Figure 6.1.

probGraphicalModels.py — (continued)

53 # Bayesian network report of leaving example from

54 # Poole and Mackworth, Artificial Intelligence, 2010 [http://artint.info](http://artint.info/)

55 # This is Example 6.10 (page 236) shown in Figure 6.1

56

57 Al = Variable("Alarm", boolean)

58 Fi = Variable("Fire", boolean)

##### Variable Elimination 145

59 Le = Variable("Leaving", boolean)

60 Re = Variable("Report", boolean)

61 Sm = Variable("Smoke", boolean)

62 Ta = Variable("Tamper", boolean)

63

64 f\_ta = Prob(Ta,[],[0.98,0.02])

65 f\_fi = Prob(Fi,[],[0.99,0.01])

66 f\_sm = Prob(Sm,[Fi],[0.99,0.01,0.1,0.9])

67 f\_al = Prob(Al,[Fi,Ta],[0.9999, 0.0001, 0.15, 0.85, 0.01, 0.99, 0.5, 0.5])

68 f\_lv = Prob(Le,[Al],[0.999, 0.001, 0.12, 0.88])

69 f\_re = Prob(Re,[Le],[0.99, 0.01, 0.25, 0.75])

70

71 bn2 = Belief\_network([Al,Fi,Le,Re,Sm,Ta],[f\_ta,f\_fi,f\_sm,f\_al,f\_lv,f\_re])

#### The third Bayesian network is the sprinkler example from Pearl.

probGraphicalModels.py — (continued)

73

74 Season = Variable("Season",["summer","winter"])

75 Sprinkler = Variable("Sprinkler",["on","off"])

76 Rained = Variable("Rained",boolean)

77 Grass\_wet = Variable("Grass wet",boolean)

78 Grass\_shiny = Variable("Grass shiny",boolean)

79 Shoes\_wet = Variable("Shoes wet",boolean)

80

81 f\_season = Prob(Season,[],[0.5,0.5])

82 f\_sprinkler = Prob(Sprinkler,[Season],[0.9,0.1,0.05,0.95])

83 f\_rained = Prob(Rained,[Season],[0.7,0.3,0.2,0.8])

84 f\_wet = Prob(Grass\_wet,[Sprinkler,Rained], [1,0,0.1,0.9,0.2,0.8,0.02,0.98])

85 f\_shiny = Prob(Grass\_shiny, [Grass\_wet], [0.95,0.05,0.3,0.7])

86 f\_shoes = Prob(Shoes\_wet, [Grass\_wet], [0.92,0.08,0.35,0.65])

87

88 bn3 = Belief\_network([Season, Sprinkler, Rained, Grass\_wet, Grass\_shiny, Shoes\_wet],

89 [f\_season, f\_sprinkler, f\_rained, f\_wet, f\_shiny, f\_shoes])

## 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 observa- tions on some other variables.

probVE.py — Variable Elimination for Graphical Models

11 **from** probFactors **import** Factor, Factor\_observed, Factor\_sum, factor\_times

12 **from** probGraphicalModels **import** Graphical\_model, Inference\_method

13

14 **class** VE(Inference\_method):

15 """The class that queries Graphical Models using variable elimination.

16

17 gm is graphical model to query

18 """

19 **def** init (self,gm=None):

20 self.gm = gm

21

22 **def** query(self,var,obs={},elim\_order=None):

23 """computes P(var|obs) where

24 var is a variable

25 obs is a variable:value dictionary"""

26 **if** var **in** obs:

27 **return** [1 **if** val == obs[var] **else** 0 **for** val **in** var.domain]

28 **else**:

29 **if** elim\_order == None:

30 elim\_order = self.gm.variables

31 projFactors = [self.project\_observations(fact,obs)

32 **for** fact **in** self.gm.factors]

33 **for** v **in** elim\_order:

34 **if** v != var **and** v **not in** obs:

35 projFactors = self.eliminate\_var(projFactors,v)

36 unnorm = factor\_times(var,projFactors)

37 p\_obs=**sum**(unnorm)

38 self.display(1,"Unnormalized probs:",unnorm,"Prob obs:",p\_obs)

39 **return** {val:pr/p\_obs **for** val,pr **in zip**(var.domain, unnorm)}

#### 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)

41 **def** project\_observations(self,factor,obs):

42 """Returns the resulting factor after observing obs

43

44 obs is a dictionary of variable:value pairs.

45 """

46 **if any**((var **in** obs) **for** var **in** factor.variables):

|  |  |  |
| --- | --- | --- |
| 47 |  | # a variable in factor is observed |
| 48 |  | **return** Factor\_observed(factor,obs) |
| 49 |  | **else**: |
| 50 |  | **return** factor |
| 51 |  |  |
| 52 | **def** | eliminate\_var(self,factors,var): |
| 53 |  | """Eliminate a variable var from a list of factors. |
| 54 |  | Returns a new set of factors that has var summed out. |
| 55 |  | """ |
| 56 |  | self.display(2,"eliminating ",**str**(var)) |
| 57 |  | contains\_var = [] |
| 58 |  | not\_contains\_var = [] |
| 59 |  | **for** fac **in** factors: |
| 60 |  | **if** var **in** fac.variables: |
| 61 |  | contains\_var.append(fac) |
| 62 |  | **else**: |

63 not\_contains\_var.append(fac)

64 **if** contains\_var == []:

65 **return** factors

66 **else**:

67 newFactor = Factor\_sum(var,contains\_var)

68 self.display(2,"Multiplying:",[f.brief() **for** f **in** contains\_var])

69 self.display(2,"Creating factor:", newFactor.brief())

70 self.display(3, newFactor) # factor in detail

71 not\_contains\_var.append(newFactor)

72 **return** not\_contains\_var

73

74 **from** probGraphicalModels **import** bn1, A,B,C

75 bn1v = VE(bn1)

76 ## bn1v.query(A,{})

77 ## bn1v.query(C,{})

78 ## Inference\_method.max\_display\_level = 3 # show more detail in displaying

79 ## Inference\_method.max\_display\_level = 1 # show less detail in displaying

80 ## bn1v.query(A,{C:True})

81 ## bn1v.query(B,{A:True,C:False})

82

83 **from** probGraphicalModels **import** bn2,Al,Fi,Le,Re,Sm,Ta

84 bn2v = VE(bn2) # answers queries using variable elimination

85 ## bn2v.query(Ta,{})

86 ## Inference\_method.max\_display\_level = 0 # show no detail in displaying

87 ## bn2v.query(Le,{})

88 ## bn2v.query(Ta,{},elim\_order=[Sm,Re,Le,Al,Fi])

89 ## bn2v.query(Ta,{Re:True})

90 ## bn2v.query(Ta,{Re:True,Sm:False})

91

92 **from** probGraphicalModels **import** bn3, Season, Sprinkler, Rained, Grass\_wet, Grass\_shiny, Shoes\_wet

93 bn3v = VE(bn3)

94 ## bn3v.query(Shoes\_wet,{})

95 ## bn3v.query(Shoes\_wet,{Rained:True})

96 ## bn3v.query(Shoes\_wet,{Grass\_shiny:True})

97 ## bn3v.query(Shoes\_wet,{Grass\_shiny:False,Rained:True})

## Stochastic Simulation

### Sampling from a discrete distribution

#### The method *sample one* generates a single sample from a (possible unnormal- ized) distribution. *dist* is a *value* : *weight* dictionary, where *weight* 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** Inference\_method

13

14 **def** sample\_one(dist):

15 """returns the index of a single sample from normalized distribution dist."""

16 rand = random.random()\***sum**(dist.values())

17 cum = 0 # cumulative weights

18 **for** v **in** dist:

19 cum += dist[v]

20 **if** cum > rand:

21 **return** v

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

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

≥

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

probStochSim.py — (continued)

23 **def** sample\_multiple(dist, num\_samples):

24 """returns a list of num\_samples values selected using distribution 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 8.1**

What is the time and space complexity the following 4 methods to generate *n*

samples, where *m* is the length of *dist*:

* + 1. *n* calls to *sample one*
    2. *sample multiple*
    3. 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.
    4. Choose a random number in the range [*i*/*n*, (*i* + 1)/*n*) for each *i 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 num- ber of samples. It is useful to see the variability as a function of the number of samples. Try it for few samples and also for many samples.

probStochSim.py — (continued)

40 **def** test\_sampling(dist, num\_samples):

41 """Given a distribution, dist, draw num\_samples samples

42 and return the resulting counts

43 """

44 result = {v:0 **for** v **in** dist}

45 **for** v **in** sample\_multiple(dist, num\_samples):

46 result[v] += 1

47 **return** result

48

49 # try the following queries a number of times each:

50 # test\_sampling({1:1,2:2,3:3,4:4}, 100)

51 # test\_sampling({1:1,2:2,3:3,4:4}, 100000)

### Sampling Methods for Belief Network Inference

#### A *Sampling inference method* is an *Inference method*, 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 Bayesian network (and not an undirected graphical model).

probStochSim.py — (continued)

53 **class** Sampling\_inference\_method(Inference\_method):

54 """The abstract class of sampling-based belief network inference methods"""

55 **def** query(self,qvar,obs={},number\_samples=1000,sample\_order=None):

56 **raise** NotImplementedError("Sampling\_inference\_method query") # abstract

#### Some of the sampling methods require a sample order of factors represent- ing conditional probabilities, where the parents of a node must come before the node in the sample order. The following method computes such a sample ordering, and is used when the *sample order* argument is *None*.

probStochSim.py — (continued)

58 **def** select\_sample\_ordering(bn):

59 """creates a sample ordering of factors such that the parents of a node

60 are before the node.

61 raises StopIteration if there is no such ordering. This would occur in next(.).

62 """

63 sample\_order=[]

64 defined = **set**() # set of variables whose probability is defined

65 factors\_to\_sample = bn.factors.copy()

66 **while** factors\_to\_sample:

67 fac = **next**(f **for** f **in** factors\_to\_sample

68 **if all**(par **in** defined **for** par **in** f.parents))

69 factors\_to\_sample.remove(fac)

70 sample\_order.append(fac)

71 defined.add(fac.child)

72 **return** sample\_order

### Rejection Sampling

probStochSim.py — (continued)

74 **class** Rejection\_sampling(Sampling\_inference\_method):

75 """The class that queries Graphical Models using Rejection Sampling.

76

77 bn is a belief network to query

78 """

79 **def** init (self,bn=None):

80 self.bn = bn

81 self.label = "Rejection Sampling"

82

83 **def** query(self,qvar,obs={},number\_samples=1000,sample\_order=None):

84 """computes P(qvar|obs) where

85 qvar is a variable.

86 obs is a variable:value dictionary.

87 sample\_order is a list of factors where factors defining the parents

88 come before the factors for the child.

89 """

90 **if** sample\_order **is** None:

91 sample\_order = select\_sample\_ordering(self.bn)

92 self.display(2,\*[f.child **for** f **in** sample\_order],sep="\t")

93 counts = {val:0 **for** val **in** qvar.domain}

94 **for** i **in range**(number\_samples):

95 rejected = False

96 sample = {}

97 **for** fac **in** sample\_order:

98 nvar = fac.child #next variable

99 val = sample\_one(fac.cond\_dist(sample))

100 self.display(2,val,end="\t")

101 **if** nvar **in** obs **and** obs[nvar] != val:

102 rejected = True

103 self.display(2,"Rejected")

104 **break**

105 sample[nvar] = val

106 **if not** rejected:

107 counts[sample[qvar]] += 1

108 self.display(2,"Accepted")

109 tot = **sum**(counts.values())

110 **return** counts, {c:divide(v,tot) **for** (c,v) **in** counts.items()}

#### It is possible that all samples get rejected. In that case, Python would give as a arithmetic error. Instead, we implement the convention that 0/0 = 1. You need to be careful is using these numbers as probabilities.

probStochSim.py — (continued)

112 **def** divide(num,denom):

113 """returns num/denom without divide-by-zero errors.

114 defines 0/0 to be 1."""

115 **if** denom == 0:

116 **return** 1.0

117 **else**:

118 **return** num/denom

### 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.

|  |  |  |  |
| --- | --- | --- | --- |
| 120  121  122  123  124  125  126 |  | probStochSim.py — (continued)  **class** Likelihood\_weighting(Sampling\_inference\_method):  """The class that queries Graphical Models using Likelihood weighting.  bn is a belief network to query """  **def** init (self,bn=None):  self.bn = bn | |
| 127 |  |  | self.label = "Likelihood weighting" |
| 128 |  |  |  |
| 129 |  | **def** | query(self,qvar,obs={},number\_samples=1000,sample\_order=None): |
| 130 |  |  | """computes P(qvar|obs) where |
| 131 |  |  | qvar is a variable. |
| 132 |  |  | obs is a variable:value dictionary. |
| 133 |  |  | sample\_order is a list of factors where factors defining the parents |
| 134 |  |  | come before the factors for the child. |
| 135 |  |  | """ |
| 136 |  |  | **if** sample\_order **is** None: |
| 137 |  |  | sample\_order = select\_sample\_ordering(self.bn) |
| 138 |  |  | self.display(2,\*[f.child **for** f **in** sample\_order |
| 139 |  |  | **if** f.child **not in** obs],sep="\t") |
| 140 |  |  | counts = [0 **for** val **in** qvar.domain] |
| 141 |  |  | **for** i **in range**(number\_samples): |
| 142 |  |  | sample = {} |
| 143 |  |  | weight = 1.0 |
| 144 |  |  | **for** fac **in** sample\_order: |
| 145 |  |  | nvar = fac.child # next variable sampled |
| 146 |  |  | **if** nvar **in** obs: |
| 147 |  |  | sample[nvar] = obs[nvar] |
| 148 |  |  | weight \*= fac.get\_value(sample) |
| 149 |  |  | **else**: |
| 150 |  |  | val = sample\_one(fac.cond\_dist(sample)) |
| 151 |  |  | self.display(2,val,end="\t") |
| 152 |  |  | sample[nvar] = val |
| 153 |  |  | counts[sample[qvar]] += weight |

154 self.display(2,weight)

155 tot = **sum**(counts)

156 **return** counts, {c:v/tot **for** (c,v) **in** counts.items()}

**Exercise 8.2** Change this algorithm so that it does **importance sampling** using a proposal distribution. It needs *sample one* using a different distribution and then update the weight of the current sample. For testing, use a proposal distribution that only specifies probabilities for some of the variables (and the algorithm uses the probabilities for the network in other cases).

* + 1. Particle Filtering

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

probStochSim.py — (continued)

158 **class** Particle\_filtering(Sampling\_inference\_method):

159 """The class that queries Graphical Models using Particle Filtering.

160

161 bn is a belief network to query

162 """

163 **def** init (self,bn=None):

164 self.bn = bn

165 self.label = "Particle Filtering"

166

167 **def** query(self, qvar, obs={}, number\_samples=1000, sample\_order=None):

168 """computes P(qvar|obs) where

169 qvar is a variable.

170 obs is a variable:value dictionary.

171 sample\_order is a list of factors where factors defining the parents

172 come before the factors for the child.

173 """

174 **if** sample\_order **is** None:

175 sample\_order = select\_sample\_ordering(self.bn)

176 self.display(2,\*[f.child **for** f **in** sample\_order

177 **if** f.child **not in** obs],sep="\t")

178 particles = [{} **for** i **in range**(number\_samples)]

179 **for** fac **in** sample\_order:

180 nvar = fac.child # the variable sampled

181 **if** nvar **in** obs:

182 weights = {part:fac.cond\_prob(part,obs[nvar]) **for** part **in** particles}

183 particles = [p.copy **for** p **in** resample(particles, weights, number\_samples)]

184 **else**:

185 **for** part **in** particles:

186 part[nvar] = sample\_one(fac.cond\_dist(part))

187 self.display(2,part[nvar],end="\t")

188 counts = [0 **for** val **in** qvar.domain]

189 **for** part **in** particles:

190 counts[part[qvar]] += 1

191 self.display(2,weight)

192 **return** counts

Resampling

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

probStochSim.py — (continued)

194 **def** resample(particles, weights, num\_samples):

195 """returns num\_samples copies of particles resampled according to weights.

196 particles is a list of particles

197 weights is a list of positive numbers, of same length as particles

198 num\_samples is n integer

199 """

200 total = **sum**(weights)

201 rands = **sorted**(random.random()\*total **for** i **in range**(num\_samples))

202 result = []

|  |  |  |
| --- | --- | --- |
| 203 | cum = weights[0] | # cumulative sum |
| 204 | index = 0 |  |
| 205 | **for** r **in** rands: |  |
| 206 | **while** r>cum: |  |
| 207 | index += 1 |  |
| 208  209  210 | cum += weights[index] result.append(particles[index])  **return** result | |

### Examples

|  |  |  |
| --- | --- | --- |
|  | | probStochSim.py — (continued) |
| 212 |  | **from** probGraphicalModels **import** bn1, A,B,C |
| 213 |  | bn1r = Rejection\_sampling(bn1) |
| 214 |  | bn1L = Likelihood\_weighting(bn1) |
| 215 |  | ## Inference\_method.max\_display\_level = 2 # detailed tracing for all inference methods |
| 216 |  | ## bn1r.query(A,{}) |
| 217 |  | ## bn1r.query(C,{}) |
| 218 |  | ## bn1r.query(A,{C:True}) |
| 219 |  | ## bn1r.query(B,{A:True,C:False}) |
| 220 |  |  |
| 221 |  | **from** probGraphicalModels **import** bn2,Al,Fi,Le,Re,Sm,Ta |
| 222 |  | bn2r = Rejection\_sampling(bn2) # answers queries using rejection sampling |
| 223 |  | bn2L = Likelihood\_weighting(bn2) # answers queries using rejection sampling |
| 224 |  | bn2p = Particle\_filtering(bn2) # answers queries using particle filtering |
| 225 |  | ## bn2r.query(Ta,{}) |
| 226 |  | ## bn2r.query(Ta,{}) |
| 227 |  | ## bn2r.query(Ta,{Re:True}) |
| 228 |  | ## Inference\_method.max\_display\_level = 0 # no detailed tracing for all inference methods |

229 ## bn2r.query(Ta,{Re:True},number\_samples=100000)

230 ## bn2r.query(Ta,{Re:True,Sm:False})

231 ## bn2r.query(Ta,{Re:True,Sm:False},number\_samples=100)

232

233 ## bn2L.query(Ta,{Re:True,Sm:False},number\_samples=100)

234 ## bn2L.query(Ta,{Re:True,Sm:False},number\_samples=100)

235

236

237 **from** probGraphicalModels **import** bn3,Season, Sprinkler

238 **from** probGraphicalModels **import** Rained, Grass\_wet, Grass\_shiny, Shoes\_wet

239 bn3r = Rejection\_sampling(bn3) # answers queries using rejection sampling

240 bn3L = Likelihood\_weighting(bn3) # answers queries using rejection sampling

241 bn3p = Particle\_filtering(bn3) # answers queries using particle filtering

242 #bn3r.query(Shoes\_wet,{Grass\_shiny:True,Rained:True})

243 #bn3L.query(Shoes\_wet,{Grass\_shiny:True,Rained:True})

244 #bn3p.query(Shoes\_wet,{Grass\_shiny:True,Rained:True})

**Exercise 8.3** This code keeps regenerating the distribution of a variable given its parents. Implement one or both of the following, and compare them to the original. Make *cond dist* return a slice that corresponds to the distribution, and then use the slice instead of the dictionary (a list slice does not generate new data structures). Make *cond dist* remember values it has already computed, and only return these.

* + 1. Plotting Behaviour of Stochastic Simulators

The stochastic simulation runs can give different answers each time they are run. For the algorithms that give the same answer in the limit as the number of samples approaches infinity (as do all of these algorithms), the algorithms can be compared by comparing the accuracy for multiple runs. Summary statistics like the variance may provide some information, but the assumptions behind the variance being appropriate (namely that the distribution is approximately Gaussian) may not hold for cases where the predictions are bounded and often skewed.

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

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

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

probStochSim.py — (continued)

246 **import** matplotlib.pyplot as plt

*8.6. Markov Chain Monte Carlo* 155

247

248 **def** plot\_stats(method, what, qvar, obs, number\_samples=100, number\_runs=1000):

249 """Plots a cumulative distribution of the prediction of the model.

250 method is a Sampling\_inference\_method (that implements appropriate query(.))

251 what is either "prob\_ev" or the value of qvar to plot

252 qvar is the query variable

253 obs is the variable:value dictionary representing the observations

254 number\_samples is the number of samples for each run

255 number\_iterations is the number of runs that are plotted

256 """

257 plt.ion()

258 plt.xlabel("value")

259 plt.ylabel("Cumulative Number")

260 Inference\_method.max\_display\_level, prev\_max\_display\_level = 0, Inference\_method.max\_display\_l

261 answers = [method.query(qvar,obs,number\_samples=number\_samples)

262 **for** i **in range**(number\_runs)]

263 **if** what == "prob\_ev":

264 values = [**sum**(ans)/number\_samples **for** ans **in** answers]

265 label = method.label+"(prob of evidence)"

266 **else**:

267 values = [divide(ans[qvar.val\_to\_index[what]],**sum**(ans)) **for** ans **in** answers]

268 label = method.label+" ("+**str**(qvar)+"="+**str**(what)+")"

269 values.sort()

270 plt.plot(values,**range**(number\_runs),label=label)

271 plt.legend(loc="upper left")

272 plt.draw()

273 Inference\_method.max\_display\_level = prev\_max\_display\_level # restore display level

274

275

276 # plot\_stats(bn2r,False,Ta,{Re:True,Sm:False},number\_samples=1000, number\_runs=1000)

277 # plot\_stats(bn2L,False,Ta,{Re:True,Sm:False},number\_samples=1000, number\_runs=1000)

278 # plot\_stats(bn2r,False,Ta,{Re:True,Sm:False},number\_samples=100, number\_runs=1000)

279 # plot\_stats(bn2L,False,Ta,{Re:True,Sm:False},number\_samples=100, number\_runs=1000)

280 # plot\_stats(bn3r,True,Shoes\_wet,{Grass\_shiny:True,Rained:True},number\_samples=1000)

281 # plot\_stats(bn3L,True,Shoes\_wet,{Grass\_shiny:True,Rained:True},number\_samples=1000)

282 # plot\_stats(bn2r,"prob\_ev",Ta,{Re:True,Sm:False},number\_samples=1000, number\_runs=1000)

283 # plot\_stats(bn2L,"prob\_ev",Ta,{Re:True,Sm:False},number\_samples=1000, number\_runs=1000)

## Markov Chain Monte Carlo

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

probMCMC.py — Markov Chain Monte Carlo (Gibbs sampling)

11 **import** random

12 **from** probGraphicalModels **import** Inference\_method

13

14 **from** probStochSim **import** sample\_one, Sampling\_inference\_method

15

16 **class** Gibbs\_sampling(Sampling\_inference\_method):

17 """The class that queries Graphical Models using Gibbs Sampling.

18

19 bn is a graphical model (e.g., a belief network) to query

20 """

21 **def** init (self,bn=None):

22 self.bn = bn

23 self.label = "Gibbs Sampling"

24

25 **def** query(self, qvar, obs={}, number\_samples=1000, burn\_in=100, sample\_order=None):

26 """computes P(qvar|obs) where

27 qvar is a variable.

28 obs is a variable:value dictionary.

29 sample\_order is a list of non-observed variables in order.

30 """

31 counts = {val:0 **for** val **in** qvar.domain}

32 **if** sample\_order **is not** None:

33 variables = sample\_order

34 **else**:

35 variables = [v **for** v **in** self.bn.variables **if** v **not in** obs]

36 var\_to\_factors = {v:**set**() **for** v **in** self.bn.variables}

37 **for** fac **in** self.bn.factors:

38 **for** var **in** fac.variables:

39 var\_to\_factors[var].add(fac)

40 sample = {var:random.choice(var.domain) **for** var **in** variables}

41 self.display(2,"Sample:",sample)

42 sample.update(obs)

43 **for** i **in range**(burn\_in + number\_samples):

44 **if** sample\_order == None:

45 random.shuffle(variables)

46 **for** var **in** variables:

47 # get probability distribution of var given its neighbours

48 vardist = {val:1 **for** val **in** var.domain}

49 **for** val **in** var.domain:

50 sample[var] = val

51 **for** fac **in** var\_to\_factors[var]: # Markov blanket

52 vardist[val] \*= fac.get\_value(sample)

53 sample[var] = sample\_one(vardist)

54 **if** i >= burn\_in:

55 counts[sample[qvar]] +=1

56 tot = **sum**(counts.values())

57 **return** counts, {c:v/tot **for** (c,v) **in** counts.items()}

58

59 **from** probGraphicalModels **import** bn1, A,B,C

60 bn1g = Gibbs\_sampling(bn1)

61 ## Inference\_method.max\_display\_level = 2 # detailed tracing for all inference methods

62 bn1g.query(A,{})

63 ## bn1g.query(C,{})

64 ## bn1g.query(A,{C:True})

65 ## bn1g.query(B,{A:True,C:False})

66

67 **from** probGraphicalModels **import** bn2,Al,Fi,Le,Re,Sm,Ta

68 bn2g = Gibbs\_sampling(bn2)

69 ## bn2g.query(Ta,{Re:True},number\_samples=100000)

**Exercise 8.4** Change the code so that it can have multiple query variables. Make the list of query variable be an input to the algorithm, so that the default value is the list of all non-observed variables.

**Exercise 8.5** In this algorithm, explain where it computes the probability of a variable given its Markov blanket. Instead of returning the average of the samples for the query variable, it is possible to return the average estimate of the probabil- ity 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?

## Hidden Markov Models

This code for hidden Markov models is independent of the graphical mod- els code, to keep it simple. Section [8.8](#_bookmark162) gives code that models hidden Markov models, and more generally, dynamic belief networks, using the graphical mod- els code.

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

probHMM.py — Hidden Markov Model

11 **import** random

12 **from** probStochSim **import** sample\_one, sample\_multiple

13

14 **class** HMM(**object**):

15 **def** init (self, states, obsvars,pobs,trans,indist):

16 """A hidden Markov model.

17 states - set of states

18 obsvars - set of observation variables

19 pobs - probability of observations, pobs[i][s] is P(Obs\_i=True | State=s)

20 trans - transition probability - trans[i][j] gives P(State=j | State=i)

21 indist - initial distribution - indist[s] is P(State\_0 = s)

22 """

23 self.states = states

24 self.obsvars = obsvars

25 self.pobs = pobs

26 self.trans = trans

27 self.indist = indist

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

probHMM.py — (continued)

29 # state

30 # 0=middle, 1,2,3 are corners

31 states1 = {'middle', 'c1', 'c2', 'c3'} # states

32 obs1 = {'m1','m2','m3'} # microphones

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

0.1. If the animal is in the middle, it will be detected by each microphone with a probability of 0.4.

probHMM.py — (continued)

34 # pobs gives the observation model:

35 #pobs[mi][state] is P(mi=on | state)

36 closeMic=0.6; farMic=0.1; midMic=0.4

37 pobs1 = {'m1':{'middle':midMic, 'c1':closeMic, 'c2':farMic, 'c3':farMic}, # mic 1

38 'm2':{'middle':midMic, 'c1':farMic, 'c2':closeMic, 'c3':farMic}, # mic 2

39 'm3':{'middle':midMic, 'c1':farMic, 'c2':farMic, 'c3':closeMic}} # mic 3

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

probHMM.py — (continued)

41 # trans specifies the dynamics

42 # trans[i] is the distribution over states resulting from state i

43 # trans[i][j] gives P(S=j | S=i)

44 sm=0.7; mmc=0.1 # transition probabilities when in middle

45 sc=0.8; mcm=0.1; mcc=0.05 # transition probabilities when in a corner

46 trans1 = {'middle':{'middle':sm, 'c1':mmc, 'c2':mmc, 'c3':mmc}, # was in middle

47 'c1':{'middle':mcm, 'c1':sc, 'c2':mcc, 'c3':mcc}, # was in corner 1

48 'c2':{'middle':mcm, 'c1':mcc, 'c2':sc, 'c3':mcc}, # was in corner 2

49 'c3':{'middle':mcm, 'c1':mcc, 'c2':mcc, 'c3':sc}} # was in corner 3

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

probHMM.py — (continued)

51 # initially we have a uniform distribution over the animal's state

52 indist1 = {st:1.0/**len**(states1) **for** st **in** states1}

53

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

* + 1. Exact Filtering for HMMs

#### A *HMM VE filter* has a current state distribution which can be updated by ob- serving or by advancing to the next time.

probHMM.py — (continued)

56 **from** display **import** Displayable

57

58 **class** HMM\_VE\_filter(Displayable):

59 **def** init (self,hmm):

60 self.hmm = hmm

61 self.state\_dist = hmm.indist

62

63 **def filter**(self, obsseq):

64 """updates and returns the state distribution following the sequence of

65 observations in obsseq using variable elimination.

66

67 Note that it first advances time.

68 This is what is required if it is called sequentially.

69 If that is not what is wanted initially, do an observe first.

70 """

71 **for** obs **in** obsseq:

72 self.advance() # advance time

73 self.observe(obs) # observe

74 **return** self.state\_dist

75

76 **def** observe(self, obs):

77 """updates state conditioned on observations.

78 obs is a list of values for each observation variable"""

79 **for** i **in** self.hmm.obsvars:

80 self.state\_dist = {st:self.state\_dist[st]\*(self.hmm.pobs[i][st]

81 **if** obs[i] **else** (1-self.hmm.pobs[i][st]))

82 **for** st **in** self.hmm.states}

83 norm = **sum**(self.state\_dist.values()) # normalizing constant

84 self.state\_dist = {st:self.state\_dist[st]/norm **for** st **in** self.hmm.states}

85 self.display(2,"After observing",obs,"state distribution:",self.state\_dist)

86

87 **def** advance(self):

88 """advance to the next time"""

89 nextstate = {st:0.0 **for** st **in** self.hmm.states} # distribution over next states

90 **for** j **in** self.hmm.states: # j ranges over next states

91 **for** i **in** self.hmm.states: # i ranges over previous states

92 nextstate[j] += self.hmm.trans[i][j]\*self.state\_dist[i]

93 self.state\_dist = nextstate

#### The following are some queries for *hmm*1.

probHMM.py — (continued)

95 hmm1f1 = HMM\_VE\_filter(hmm1)

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

97 ## HMM\_VE\_filter.max\_display\_level = 2 # show more detail in displaying

98 # hmm1f2 = HMM\_VE\_filter(hmm1)

99 # hmm1f2.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':1, 'm3':0}, {'m1':1, 'm2':0, 'm3':0},

100 # {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0},

101 # {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':1}, {'m1':0, 'm2':0, 'm3':1},

102 # {'m1':0, 'm2':0, 'm3':1}])

103 # hmm1f3 = HMM\_VE\_filter(hmm1)

104 # hmm1f3.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':1, 'm2':0, 'm3':0}, {'m1':

105

106 # How do the following differ in the resulting state distribution?

107 # Note they start the same, but have different initial observations.

108 ## HMM\_VE\_filter.max\_display\_level = 1 # show less detail in displaying

109 # for i in range(100): hmm1f1.advance()

110 # hmm1f1.state\_dist

111 # for i in range(100): hmm1f3.advance()

112 # hmm1f3.state\_dist

**Exercise 8.6** The localization example in the book is a controlled HMM, where there is a given action at each time and the transition depends on the action. Change the code to allow for controlled HMMs. Hint: the action only influences the state transition.

**Exercise 8.7** The representation assumes that there are a list of Boolean obser- vations. 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.

* + 1. Particle Filtering for HMMs

In this implementation a particle is just a state. If you want to do some form of smooting, 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)

113 **from** display **import** Displayable

114 **from** probStochSim **import** resample

115

116 **class** HMM\_particle\_filter(Displayable):

117 **def** init (self,hmm,number\_particles=1000):

118 self.hmm = hmm

119 self.particles = [sample\_one(hmm.indist)

120 **for** i **in range**(number\_particles)]

121 self.weights = [1 **for** i **in range**(number\_particles)]

122

123 **def filter**(self, obsseq):

124 """returns the state distribution following the sequence of

125 observations in obsseq using particle filtering.

126

127 Note that it first advances time.

128 This is what is required if it is called after previous filtering.

129 If that is not what is wanted initially, do an observe first.

130 """

131 **for** obs **in** obsseq:

132 self.advance() # advance time

133 self.observe(obs) # observe

|  |  |  |
| --- | --- | --- |
| 134  135  136 | self.resample\_particles()  self.display(2,"After observing", **str**(obs),  "state distribution:", self.histogram(self.particles)) | |
| 137 |  | self.display(1,"Final state distribution:", self.histogram(self.particles)) |
| 138 |  | **return** self.histogram(self.particles) |
| 139 |  |  |
| 140 | **def** | advance(self): |
| 141 |  | """advance to the next time. |
| 142 |  | This assumes that all of the weights are 1.""" |
| 143 |  | self.particles = [sample\_one(self.hmm.trans[st]) |
| 144 |  | **for** st **in** self.particles] |
| 145 |  |  |
| 146 | **def** | observe(self, obs): |
| 147 |  | """reweight the particles to incorporate observations obs""" |
| 148 |  | **for** i **in range**(**len**(self.particles)): |
| 149 |  | **for** obv **in** obs: |
| 150 |  | **if** obs[obv]: |
| 151  152  153 | self.weights[i] \*= self.hmm.pobs[obv][self.particles[i]]  **else**:  self.weights[i] \*= 1-self.hmm.pobs[obv][self.particles[i]] | |
| 154 |  |  |
| 155 | **def** | histogram(self, particles): |
| 156 |  | """returns list of the probability of each state as represented by |
| 157 |  | the particles""" |
| 158 |  | tot=0 |
| 159 |  | hist = {st: 0.0 **for** st **in** self.hmm.states} |
| 160 |  | **for** (st,wt) **in zip**(self.particles,self.weights): |
| 161 |  | hist[st]+=wt |
| 162 |  | tot += wt |
| 163 |  | **return** {st:hist[st]/tot **for** st **in** hist} |
| 164 |  |  |
| 165 | **def** | resample\_particles(self): |
| 166 |  | """resamples to give a new set of particles.""" |
| 167 |  | self.particles = resample(self.particles, self.weights, **len**(self.particles)) |
| 168 |  | self.weights = [1] \* **len**(self.particles) |

#### The following are some queries for *hmm*1.

|  |  |  |
| --- | --- | --- |
|  | | probHMM.py — (continued) |
| 170 |  | hmm1pf1 = HMM\_particle\_filter(hmm1) |
| 171 |  | # HMM\_particle\_filter.max\_display\_level = 2 # show each step |
| 172 |  | # hmm1pf1.filter([{'m1':0, 'm2':1, 'm3':1}, {'m1':1, 'm2':0, 'm3':1}]) |
| 173 |  | # hmm1pf2 = HMM\_particle\_filter(hmm1) |
| 174 |  | # hmm1pf2.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':1, 'm3':0}, {'m1':1, 'm2':0, 'm3':0}, |
| 175 |  | # {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, |
| 176 |  | # {'m1':0, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':1}, {'m1':0, 'm2':0, 'm3':1}, |
| 177 |  | # {'m1':0, 'm2':0, 'm3':1}]) |
| 178  179 |  | # hmm1pf3 = HMM\_particle\_filter(hmm1)  # hmm1pf3.filter([{'m1':1, 'm2':0, 'm3':0}, {'m1':0, 'm2':0, 'm3':0}, {'m1':1, 'm2':0, 'm3':0}, |

{'

**Exercise 8.8** A form of importance sampling can be obtained by not resampling.

Is it better or worse than particle filtering? Hint: you need to think about how they can be compared. Is the comparison different if there are more states than particles?

**Exercise 8.9** Extend the particle filtering code to continuous variables and ob- servations. 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.

* + 1. Generating Examples

The following code is useful for generating examples.

probHMM.py — (continued)

181 **def** simulate(hmm,horizon):

182 """returns a pair of (state sequence, observation sequence) of length horizon.

183 for each time t, the agent is in state\_sequence[t] and

184 observes observation\_sequence[t]

185 """

186 state = sample\_one(hmm.indist)

187 obsseq=[]

188 stateseq=[]

189 **for** time **in range**(horizon):

190 stateseq.append(state)

191 newobs = {obs:sample\_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})

192 **for** obs **in** hmm.obsvars}

193 obsseq.append(newobs)

194 state = sample\_one(hmm.trans[state])

195 **return** stateseq,obsseq

196

197 **def** simobs(hmm,stateseq):

198 """returns observation sequence for the state sequence"""

199 obsseq=[]

200 **for** state **in** stateseq:

201 newobs = {obs:sample\_one({0:1-hmm.pobs[obs][state],1:hmm.pobs[obs][state]})

202 **for** obs **in** hmm.obsvars}

203 obsseq.append(newobs)

|  |  |  |
| --- | --- | --- |
| 204 |  | **return** obsseq |
| 205 |  |  |
| 206 | **def** | create\_eg(hmm,n): |
| 207 |  | """Create an annotated example for horizon n""" |
| 208 |  | seq,obs = simulate(hmm,n) |
| 209 |  | **print**("True state sequence:",seq) |
| 210 |  | **print**("Sequence of observations:\n",obs) |
| 211 |  | hmmfilter = HMM\_VE\_filter(hmm) |
| 212 |  | dist = hmmfilter.**filter**(obs) |
| 213 |  | **print**("Resulting distribution over states:\n",dist) |

## Dynamic Belief Networks

#### A dynamic belief network consists of:

* + - A set of features. A variable is a feature-time pair.
    - An initial distribution over the features at time 0. This is a belief network with all variables being time 0 variables.
    - A specification of the dynamics. Here we define the how the variables one time depend on variables at that time and the previous time, in such a way that the graph is acyclic.

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 net- work 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. However, the unrolled Bayesian network may be very large. We also need to construct multiple copies of each feature.
    - Just representing the variables “now”. In this approach we can observe and query the current variables. We can them move to the next time. This does not allow for arbitrary historical queries (about the past or the future), but can be much simpler.

Here we will implement the second of these.

probDBN.py — Dynamic belief networks

11 **from** probVariables **import** Variable

12 **from** probGraphicalModels **import** Graphical\_model

13 **from** probFactors **import** Prob, Factor\_rename

14 **from** probVE **import** VE

15 **from** display **import** Displayable

16

17 **class** DBN\_variable(Variable):

18 """A random variable that incorporates

19

20 A variable can have both a name and an index. The index defaults to 1.

21 Equality is true if they are both the name and the index are the same."""

22 **def** init (self,name,domain=[False,True],index=1):

23 Variable. init (self,name,domain)

24 self.index = index

25 self.previous = None

26

27 **def** lt (self,other):

28 **if** self.name != other.name:

29 **return** self.name<other.name

30 **else**:

31 **return** self.index<other.index

32

33 **def** gt (self,other):

34 **return** other<self

35

36 **def** str (self):

37 # if self.index==1:

38 # return self.name

39 # else:

40 **return** self.name+"\_"+**str**(self.index)

41

42 repr = str

43

44 **def** variable\_pair(name,domain=[False,True]):

45 """returns a variable and its predecessor. This is used to define 2-stage DBNs

46

47 If the name is X, it returns the pair of variables X0,X"""

48 var = DBN\_variable(name,domain)

49 var0 = DBN\_variable(name,domain,index=0)

50 var.previous = var0

51 **return** var0, var

probDBN.py — (continued)

53 **class** DBN(Displayable):

54 """The class of stationary Dynamic Bayesian networks.

55

56 \* vars1 is a list of current variables (each must have

57 previous variable).

58 \* transition\_factors is a list of factors for P(X|parents) where X

59 is a current variable and parents is a list of current or previous variables.

60 \* init\_factors is a list of factors for P(X|parents) where X is a

61 current variable and parents can only include current variables

62 The graph of transition factors + init factors must be acyclic.

63

64 """

65 **def** init (self,vars1, transition\_factors=None, init\_factors=None):

66 self.vars1 = vars1

67 self.vars0 = [v.previous **for** v **in** vars1]

68 self.transition\_factors = transition\_factors

69 self.init\_factors = init\_factors

70 self.var\_index = {} # var\_index[v] is the index of variable v

71 **for** i,v **in enumerate**(vars1):

72 self.var\_index[v]=i

#### Here is a 3 variable DBN:

probDBN.py — (continued)

74 A0,A1 = variable\_pair("A")

75 B0,B1 = variable\_pair("B")

76 C0,C1 = variable\_pair("C")

77

78 # dynamics

79 pc = Prob(C1,[B1,C0],[0.03,0.97,0.38,0.62,0.23,0.77,0.78,0.22])

80 pb = Prob(B1,[A0,A1],[0.5,0.5,0.77,0.23,0.4,0.6,0.83,0.17])

81 pa = Prob(A1,[A0,B0],[0.1,0.9,0.65,0.35,0.3,0.7,0.8,0.2])

82

83 # initial distribution

84 pa0 = Prob(A1,[],[0.9,0.1])

85 pb0 = Prob(B1,[A1],[0.3,0.7,0.8,0.2])

86 pc0 = Prob(C1,[],[0.2,0.8])

87

88 dbn1 = DBN([A1,B1,C1],[pa,pb,pc],[pa0,pb0,pc0])

#### Here is the animal example

probDBN.py — (continued)

90 **from** probHMM **import** closeMic, farMic, midMic, sm, mmc, sc, mcm, mcc

91

92 Pos\_0,Pos\_1 = variable\_pair("Position",domain=[0,1,2,3])

93 Mic1\_0,Mic1\_1 = variable\_pair("Mic1")

94 Mic2\_0,Mic2\_1 = variable\_pair("Mic2")

95 Mic3\_0,Mic3\_1 = variable\_pair("Mic3")

96

97 # conditional probabilities - see hmm for the values of sm,mmc, etc

98 ppos = Prob(Pos\_1, [Pos\_0],

99 [sm, mmc, mmc, mmc, #was in middle

100 mcm, sc, mcc, mcc, #was in corner 1

101 mcm, mcc, sc, mcc, #was in corner 2

102 mcm, mcc, mcc, sc]) #was in corner 3

103 pm1 = Prob(Mic1\_1, [Pos\_1], [1-midMic, midMic, 1-closeMic, closeMic,

104 1-farMic, farMic, 1-farMic, farMic])

105 pm2 = Prob(Mic2\_1, [Pos\_1], [1-midMic, midMic, 1-farMic, farMic,

106 1-closeMic, closeMic, 1-farMic, farMic])

107 pm3 = Prob(Mic3\_1, [Pos\_1], [1-midMic, midMic, 1-farMic, farMic,

108 1-farMic, farMic, 1-closeMic, closeMic])

109 ipos = Prob(Pos\_1,[], [0.25, 0.25, 0.25, 0.25])

110 dbn\_an =DBN([Pos\_1,Mic1\_1,Mic2\_1,Mic3\_1],

111 [ppos, pm1, pm2, pm3],

112 [ipos, pm1, pm2, pm3])

|  |  |  |
| --- | --- | --- |
|  |  | probDBN.py — (continued) |
| 114 |  | **class** DBN\_VE\_filter(VE): |
| 115 |  | **def** init (self,dbn): |
| 116 |  | self.dbn = dbn |
| 117 |  | self.current\_factors = dbn.init\_factors |
| 118 |  | self.current\_obs = {} |
| 119 |  |  |
| 120 |  | **def** observe(self, obs): |
| 121 |  | """updates the current observations with obs. |
| 122 |  | obs is a variable:value dictionary where variable is a current |
| 123 |  | variable. |

124 """

125 **assert all**(self.current\_obs[var]==obs[var] **for** var **in** obs

126 **if** var **in** self.current\_obs),"inconsistent current observations"

127 self.current\_obs.update(obs)

128

129 **def** query(self,var):

130 """returns the posterior probability of current variable var"""

131 **return** VE(Graphical\_model(self.dbn.vars1,self.current\_factors)).query(var,self.current\_obs)

132

133 **def** advance(self):

134 """advance to the next time"""

135 prev\_factors = [self.make\_previous(fac) **for** fac **in** self.current\_factors]

136 prev\_obs = {var.previous:val **for** var,val **in** self.current\_obs.items()}

137 two\_stage\_factors = prev\_factors + self.dbn.transition\_factors

138 self.current\_factors = self.elim\_vars(two\_stage\_factors,self.dbn.vars0,prev\_obs)

139 self.current\_obs = {}

140

141 **def** make\_previous(self,fac):

142 """Creates new factor from fac where the current variables in fac

143 are renamed to previous variables.

144 """

145 **return** Factor\_rename(fac, {var.previous:var **for** var **in** fac.variables})

146

147 **def** elim\_vars(self,factors, **vars**, obs):

148 **for** var **in vars**:

149 **if** var **in** obs:

150 factors = [self.project\_observations(fac,obs) **for** fac **in** factors]

151 **else**:

152 factors = self.eliminate\_var(factors, var)

153 **return** factors

#### Example queries:

probDBN.py — (continued)

155 df = DBN\_VE\_filter(dbn1)

156 #df.observe({B1:True}); df.advance(); df.observe({C1:False})

157 #df.query(B1)

158 #df.advance()

159 #df.query(B1)

160 dfa = DBN\_VE\_filter(dbn\_an)

161 # dfa.observe({Mic1\_1:0, Mic2\_1:1, Mic3\_1:1})

162 # dfa.advance()

163 # dfa.observe({Mic1\_1:1, Mic2\_1:0, Mic3\_1:1})

164 # dfa.query(Pos\_1)

# Chapter 9

Planning with Uncertainty

## 9.1 Decision Networks

#### The decision network code builds on the representation for belief networks of Chapter [8.](#_bookmark133)

We first allow for factors that define the utility. Here the utility is a function of the variables in *vars*, and the table is a list that enumerates the values as in Section [8.2.](#_bookmark138)

decnNetworks.py — Representations for Decision Networks

11 **from** probGraphicalModels **import** Graphical\_model

12 **from** probFactors **import** Factor\_stored

13 **from** probVariables **import** Variable

14 **from** probFactors **import** Prob

15

16 **class** Utility(Factor\_stored):

17 """A factor defined by a utility"""

18 **def** init (self,**vars**,table):

19 """Creates a factor on vars from the table.

20 The table is ordered according to vars.

21 """

22 Factor\_stored. init (self,**vars**,table)

23 **assert** self.size==**len**(table),"Table size incorrect "+**str**(self)

#### A decision variable is a like a random variable with a string name, and a do- main, 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.

decnNetworks.py — (continued)

25 **class** DecisionVariable(Variable):

26 **def** init (self,name,domain,parents):

#### 167

27 Variable. init (self,name,domain)

28 self.parents = parents

29 self.all\_vars = **set**(parents) | {self}

#### A decision network is a graphical model where the variables can be random variables or decision variables. In the factors we assume there is one utility factor.

decnNetworks.py — (continued)

31 **class** DecisionNetwork(Graphical\_model):

32 **def** init (self,**vars**=None,factors=None):

33 """vars is a list of variables

34 factors is a list of factors (instances of Prob and Utility)

35 """

36 Graphical\_model. init (self,**vars**,factors)

#### VE DN is variable elimination for decision networks. The method *optimize* is used to optimize all the decisions. Note that *optimize* requires a legal emimina- tion ordering of the random and decision variables, otherwise it will give an exception. (A decision node can only be maximized if the variables that are not its parents have already been eliminated.)

decnNetworks.py — (continued)

38 **from** probFactors **import** factor\_times, Factor\_stored

39 **from** probVE **import** VE

40

41 **class** VE\_DN(VE):

42 """Variable Elimination for Decision Networks"""

43 **def** init (self,dn=None):

44 """dn is a decision network"""

45 VE. init (self,dn)

46 self.dn = dn

47

48 **def** optimize(self,elim\_order=None,obs={}):

49 **if** elim\_order == None:

50 elim\_order = self.gm.variables

51 policy = []

52 proj\_factors = [self.project\_observations(fact,obs)

53 **for** fact **in** self.dn.factors]

54 **for** v **in** elim\_order:

55 **if isinstance**(v,DecisionVariable):

56 to\_max = [fac **for** fac **in** proj\_factors

57 **if** v **in** fac.variables **and set**(fac.variables) <= v.all\_vars]

58 **assert len**(to\_max)==1, "illegal variable order "+**str**(elim\_order)+" at "+**str**(v)

59 newFac = Factor\_max(v, to\_max[0])

60 policy.append(newFac.decision\_fun)

61 proj\_factors = [fac **for** fac **in** proj\_factors **if** fac **is not** to\_max[0]]+[newFac]

62 self.display(2,"maximizing",v,"resulting factor",newFac.brief() )

63 self.display(3,newFac)

64 **else**:

65 proj\_factors = self.eliminate\_var(proj\_factors, v)

66 **assert len**(proj\_factors)==1,"Should there be only one element of proj\_factors?"

67 value = proj\_factors[0].get\_value({})

68 **return** value,policy

decnNetworks.py — (continued)

70 **class** Factor\_max(Factor\_stored):

71 """A factor obtained by maximizing a variable in a factor.

72 Also builds a decision\_function. This is based on Factor\_sum.

73 """

74

75 **def** init (self, dvar, factor):

76 """dvar is a decision variable.

77 factor is a factor that contains dvar and only parents of dvar

78 """

79 self.dvar = dvar

80 self.factor = factor

81 **vars** = [v **for** v **in** factor.variables **if** v **is not** dvar]

82 Factor\_stored. init (self,**vars**,None)

83 self.values = [None]\*self.size

84 self.decision\_fun = Factor\_DF(dvar,**vars**,[None]\*self.size)

85

86 **def** get\_value(self,assignment):

87 """lazy implementation: if saved, return saved value, else compute it"""

88 index = self.assignment\_to\_index(assignment)

89 **if** self.values[index]:

90 **return** self.values[index]

91 **else**:

92 max\_val = **float**("-inf") # -infinity

93 new\_asst = assignment.copy()

94 **for** elt **in** self.dvar.domain:

95 new\_asst[self.dvar] = elt

96 fac\_val = self.factor.get\_value(new\_asst)

97 **if** fac\_val>max\_val:

98 max\_val = fac\_val

99 best\_elt = elt

100 self.values[index] = max\_val

101 self.decision\_fun.values[index] = best\_elt

102 **return** max\_val

#### A decision function is a stored factor.

decnNetworks.py — (continued)

104 **class** Factor\_DF(Factor\_stored):

105 """A decision function"""

106 **def** init (self,dvar, **vars**, values):

107 Factor\_stored. init (self,**vars**,values)

108 self.dvar = dvar

109 self.name = **str**(dvar) # Used in printing

#### The fire decision network of Figure [9.1](#_bookmark171) is represented as:



*Tampering*

*Fire*

*Utility*

*Alarm*

*Smoke*

*Leaving*

*See\_smoke*

*Report*

*Call*

*Check\_smoke*

Figure 9.1: Fire Decision Network

decnNetworks.py — (continued)

111 boolean = [False, True]

112 Al = Variable("Alarm", boolean)

113 Fi = Variable("Fire", boolean)

114 Le = Variable("Leaving", boolean)

115 Re = Variable("Report", boolean)

116 Sm = Variable("Smoke", boolean)

117 Ta = Variable("Tamper", boolean)

118 SS = Variable("See Sm", boolean)

119 CS = DecisionVariable("Ch Sm", boolean,{Re})

120 Call = DecisionVariable("Call", boolean,{SS,CS,Re})

121

122 f\_ta = Prob(Ta,[],[0.98,0.02])

123 f\_fi = Prob(Fi,[],[0.99,0.01])

124 f\_sm = Prob(Sm,[Fi],[0.99,0.01,0.1,0.9])

125 f\_al = Prob(Al,[Fi,Ta],[0.9999, 0.0001, 0.15, 0.85, 0.01, 0.99, 0.5, 0.5])

126 f\_lv = Prob(Le,[Al],[0.999, 0.001, 0.12, 0.88])

127 f\_re = Prob(Re,[Le],[0.99, 0.01, 0.25, 0.75])

128 f\_ss = Prob(SS,[CS,Sm],[1,0,1,0,1,0,0,1])

129

130 ut = Utility([CS,Fi,Call],[0,-200,-5000,-200,-20,-220,-5020,-220])

131

132 dnf = DecisionNetwork([Ta,Fi,Al,Le,Sm,Call,SS,CS,Re],[f\_ta,f\_fi,f\_sm,f\_al,f\_lv,f\_re,f\_ss,ut])

133 # v,p = VE\_DN(dnf).optimize()

134 # for df in p: print(df,"\n")

#### The following is the representation of the cheating decision of Figure [9.2.](#_bookmark172) Note that we keep the names of the variables short (less than 8 characters) so that tables Python prints look good.



*Watched*

*Punishment*

*Caught*1

*Caught*2

*Utility*

*Grade*1

*Grade*2

*Final Grade*

*Cheat*1

*Cheat*2

Figure 9.2: Cheating Decision Network

|  |  |  |
| --- | --- | --- |
|  | | decnNetworks.py — (continued) |
| 136 |  | grades = ["A","B","C","F"] |
| 137 |  | Wa = Variable("Watched", boolean) |
| 138 |  | CC1 = Variable("Caught1", boolean) |
| 139 |  | CC2 = Variable("Caught2", boolean) |
| 140 |  | Pun = Variable("Punish",["None","Suspension","Recorded"]) |
| 141 |  | Gr1 = Variable("Grade\_1",grades) |
| 142 |  | Gr2 = Variable("Grade\_2",grades) |
| 143 |  | GrF = Variable("Fin\_Grd",grades) |
| 144 |  | Ch1 = DecisionVariable("Cheat\_1", boolean,**set**()) #no parents |
| 145 |  | Ch2 = DecisionVariable("Cheat\_2", boolean,{Ch1,CC1}) |
| 146 |  |  |
| 147 |  | p\_wa = Prob(Wa,[],[0.7, 0.3]) |
| 148 |  | p\_cc1 = Prob(CC1,[Wa,Ch1],[1.0, 0.0, 0.9, 0.1, 1.0, 0.0, 0.5, 0.5]) |
| 149 |  | p\_cc2 = Prob(CC2,[Wa,Ch2],[1.0, 0.0, 0.9, 0.1, 1.0, 0.0, 0.5, 0.5]) |
| 150 |  | p\_pun = Prob(Pun,[CC1,CC2],[1.0, 0.0, 0.0, 0.5, 0.4, 0.1, 0.6, 0.2, 0.2, 0.2, 0.5, 0.3]) |
| 151 |  | p\_gr1 = Prob(Gr1,[Ch1], [0.2, 0.3, 0.3, 0.2, 0.5, 0.3, 0.2, 0.0]) |
| 152 |  | p\_gr2 = Prob(Gr2,[Ch2], [0.2, 0.3, 0.3, 0.2, 0.5, 0.25, 0.25, 0.0]) |
| 153 |  | p\_fg = Prob(GrF,[Gr1,Gr2], |
| 154 |  | [1.0, 0.0, 0.0, 0.0, 0.5, 0.5, 0.0, 0.0, 0.25, 0.5, 0.25, 0.0, 0.25, |
| 155 |  | 0.25, 0.25, 0.25, 0.5, 0.5, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.5, |
| 156 |  | 0.5, 0.0, 0.0, 0.25, 0.5, 0.25, 0.25, 0.5, 0.25, 0.0, 0.0, 0.5, 0.5, |
| 157 |  | 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.25, 0.75, 0.25, 0.5, 0.25, 0.0, |
| 158 |  | 0.0, 0.25, 0.5, 0.25, 0.0, 0.0, 0.25, 0.75, 0.0, 0.0, 0.0, 1.0]) |
| 159 |  | utc = Utility([Pun,GrF],[100,90,70,50,40,20,10,0,70,60,40,20]) |

160

161 cheat\_dn = DecisionNetwork([Pun,CC2,Wa,GrF,Gr2,Gr1,Ch2,CC1,Ch1],

162 [p\_wa, p\_cc1, p\_cc2, p\_pun, p\_gr1, p\_gr2,p\_fg,utc])

163

164 # VE\_DN.max\_display\_level = 3 # if you want to show lots of detail

165 # v,p = VE\_DN(cheat\_dn).optimize(); print(v)

166 # for df in p: print(df,"\n") # print decision functions

## Markov Decision Processes

#### We will represent a **Markov decision process** (**MDP**) directly, rather than using the variable elimination code, as we did for decision networks.

States and actions are represented as lists of strings. The data structures for transitions, rewards, q-values, etc., use the index of the state or the action. The names of the state with index *i* is in *states*[*i*], and the name of action with index *i* is in *actions*[*i*].

mdpProblem.py — Representations for Markov Decision Processes

11 **from** utilities **import** argmax

12

13 **class** MDP(**object**):

14 **def** init (self, states, actions, trans, reward, discount):

15 """states is a list or tuple of states.

16 actions is a list or tuple of actions

17 trans[s][a][s'] represents P(s'|a,s)

18 reward[s][a] gives the expected reward of doing a in state s

19 discount is a real in the range [0,1]

20 """

21 self.states = states

22 self.actions = actions

23 self.trans = trans

24 self.reward = reward

25 self.discount = discount

26 self.v0 = [0 **for** s **in** states] # initial value function

#### 2 state partying example:

mdpExamples.py — MDP Examples

11 **from** mdpProblem **import** MDP

12 #### Partying Decision Example ####

13

14 # States: Healthy Sick

15 # Actions: Relax Party

16

17 # trans[s][a][s'] gives P(s'|a,s)

18 # Relax Party

19 trans2 = (((0.95,0.05), (0.7, 0.3)), # Healthy

20 ((0.5,0.5), (0.1, 0.9)) # Sick

21 )

22

23 # reward[s][a] gives the expected reward of doing a in state s.

24 reward2 = ((7,10),(0,2))

25

26 healthy2 = MDP(['Healthy','Sick'], ['Relax','Party'], trans2, reward2, discount=0.8)

#### Tiny game from Example 11.7 and Figure 11.8 of Poole and Mackworth, 2010:

mdpExamples.py — (continued)

28 ## Tiny Game from Example 11.7 and Figure 11.8 of Poole and Mackworth, 2010 #

29

30 # actions up right upC left

31 transt = (((0.1,0.1,0.8,0,0,0), (0,1,0,0,0,0), (0,0,1,0,0,0), (1,0,0,0,0,0)), #s0

32 ((0.1,0.1,0,0.8,0,0), (0,1,0,0,0,0), (0,0,0,1,0,0), (1,0,0,0,0,0)),

#s1

33 ((0,0,0.1,0.1,0.8,0), (0,0,0,1,0,0), (0,0,0,0,1,0), (0,0,1,0,0,0)),

#s2

34 ((0,0,0.1,0.1,0,0.8), (0,0,0,1,0,0), (0,0,0,0,0,1), (0,0,1,0,0,0)),

#s3

35 ((0.1,0,0,0,0.8,0.1), (0,0,0,0,0,1), (0,0,0,0,1,0), (1,0,0,0,0,0)),

#s4

36 ((0,0,0,0,0.1,0.9), (0,0,0,0,0,1), (0,0,0,0,0,1), (0,0,0,0,1,0)) ) #s5

37

38 # actions up rt upC left

39 rewardt = ((-0.1, 0, -1, -1), #s0

40 (-0.1, -1, -2, 0), #s1

41 (-10, 0, -1, -100), #s2

42 (-0.1, -1, -1, 0), #s3

43 (-1, 0, -2, 10), #s4

44 (-1, -1, -2, 0)) #s5

45

46 mdpt = MDP(['s0','s1','s2','s3','s4','s5'], # states

47 ['up', 'right', 'upC', 'left'], # actions

48 transt, rewardt, discount=0.9)

## Value Iteration

#### This implements value iteration, storing *V*.

This uses indexes of the states and actions (not the names). A value function is list, *v*, such that *v*[*s*] is the value for state with index *s*. Similarly a policy *pi* is represented as a list where *pi*[*s*], where *s* is the index of a state, returns the index of the action.

mdpProblem.py — (continued)

28 **def** vi1(self,v):

29 """carry out one iteration of value iteration and

30 returns a value function (a list of a value for each state).

31 v is the previous value function.

32 """

33 **return** [**max**([self.reward[s][a]+self.discount\*product(self.trans[s][a],v)

34 **for** a **in range**(**len**(self.actions))])

35 **for** s **in range**(**len**(self.states))]

36

37 **def** vi(self,v0,n):

38 """carries out n iterations of value iteration starting with value v0.

39

40 Returns a value function

41 """

42 val = self.v0

43 **for** i **in range**(n):

44 val= self.vi1(val)

45 **return** val

46

47 **def** policy(self,v):

48 """returns an optimal policy assuming the next value function is v

49 v is a list of values for each state

50 returns a list of the indexes of optimal actions for each state

51 """

52 **return** [argmax(**enumerate**([self.reward[s][a]+self.discount\*product(self.trans[s][a],v)

53 **for** a **in range**(**len**(self.actions))]))

54 **for** s **in range**(**len**(self.states))]

55

56 **def** q(self,v):

57 """returns the one-step-lookahead q-value assuming the next value function is v

58 v is a list of values for each state

59 returns a list of q values for each state. so that q[s][a] represents Q(s,a)

60 """

61 **return** [[self.reward[s][a]+self.discount\*product(self.trans[s][a],v)

62 **for** a **in range**(**len**(self.actions))]

63 **for** s **in range**(**len**(self.states))]

mdpProblem.py — (continued)

65 **def** product(l1,l2):

66 """returns the dot product of l1 and l2"""

67 **return sum**([i1\*i2 **for** (i1,i2) **in zip**(l1,l2)])

#### The following gives a trace for the examples:

mdpExamples.py — (continued)

50 **def** trace(mdp,numiter):

51 **print**("Q values are shown as",[[st+"\_"+ac **for** ac **in** mdp.actions] **for** st **in** mdp.states])

52 **print**("One step lookahead Q-values:")

53 **print**(mdp.q(mdp.v0))

54 **print**("Values are for the states:", mdp.states)

55 **print**("One step lookahead values:")

56 **print**(mdp.vi(mdp.v0,1))

57 **print**("Two step lookahead Q-values:")

58 **print**(mdp.q(mdp.vi(mdp.v0,1)))

59 **print**("Two step lookahead values:")

60 **print**(mdp.vi(mdp.v0,2))

61 vfin = mdp.vi(mdp.v0,numiter)

62 **print**("After",numiter,"iterations, values:")

63 **print**(vfin)

64 **print**("After",numiter,"iterations, Q-values:")

65 **print**(mdp.q(vfin))

66 **print**("After",numiter,"iterations, Policy:",

67 [st+"->"+mdp.actions[act] **for** (st,act) **in zip**(mdp.states ,mdp.policy(vfin))])

68

69 # Try the following:

70 # trace(healthy2,10)

**Exercise 9.1** Implement value iteration that stores the *Q*-values rather than the

*V*-values. Does it work better than storing *V*? (What might better mean?)

**Exercise 9.2** Implement asynchronous value iteration. Try a number of differ- ent ways to choose the states and actions to update (e.g., sweeping through the state-action pairs, choosing them at random). Note that the best way may be to determine which states have had their Q-values change the most, and then update the previous ones, but that is not so straightforward to implement, because you need to find those previous states.

Chapter 10

Learning with Uncertainty

## K-means

The k-means learner maintains two lists that suffice as sufficient statistics to classify examples, and to learn the classification:

* + - *class counts* is a list such that *class counts*[*c*] is the number of examples in the training set with *class* = *c*.

#### *feature sum* is a list such that *feature sum*[*i*][*c*] is sum of the values for the *i*’th feature *i* for members of class *c*. The average value of the *i*th feature in class *i* is

*feature sum*[*i*][*c*] *class counts*[*c*]

#### The class is initialized by randomly assigning examples to classes, and updat- ing the statistics for *class counts* and *feature sum*.

learnKMeans.py — k-means learning

11 **from** learnProblem **import** Data\_set, Learner, Data\_from\_file

12 **import** random

13 **import** matplotlib.pyplot as plt

14

15 **class** K\_means\_learner(Learner):

16 **def** init (self,dataset, num\_classes):

17 self.dataset = dataset

18 self.num\_classes = num\_classes

19 self.random\_initialize()

20

21 **def** random\_initialize(self):

#### 177

22 # class\_counts[c] is the number of examples with class=c

23 self.class\_counts = [0]\*self.num\_classes

24 # feature\_sum[i][c] is the sum of the values of feature i for class c

25 self.feature\_sum = [[0]\*self.num\_classes

26 **for** feat **in** self.dataset.input\_features]

27 **for** eg **in** self.dataset.train:

28 cl = random.randrange(self.num\_classes) # assign eg to random class

29 self.class\_counts[cl] += 1

30 **for** (ind,feat) **in enumerate**(self.dataset.input\_features):

31 self.feature\_sum[ind][cl] += feat(eg)

32 self.num\_iterations = 0

33 self.display(1,"Initial class counts: ",self.class\_counts)

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

learnKMeans.py — (continued)

35 **def** distance(self,cl,eg):

36 """distance of the eg from the mean of the class"""

37 **return sum**( (self.class\_prediction(ind,cl)-feat(eg))\*\*2

38 **for** (ind,feat) **in enumerate**(self.dataset.input\_features))

39

40 **def** class\_prediction(self,feat\_ind,cl):

41 """prediction of the class cl on the feature with index feat\_ind"""

42 **if** self.class\_counts[cl] == 0:

43 **return** 0 # there are no examples so we can choose any value

44 **else**:

45 **return** self.feature\_sum[feat\_ind][cl]/self.class\_counts[cl]

46

47 **def** class\_of\_eg(self,eg):

48 """class to which eg is assigned"""

49 **return** (**min**((self.distance(cl,eg),cl)

50 **for** cl **in range**(self.num\_classes)))[1]

51 # second element of tuple, which is a class with minimum distance

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

learnKMeans.py — (continued)

53 **def** k\_means\_step(self):

54 """Updates the model with one step of k-means.

55 Returns whether the assignment is stable.

56 """

57 new\_class\_counts = [0]\*self.num\_classes

58 # feature\_sum[i][c] is the sum of the values of feature i for class c

59 new\_feature\_sum = [[0]\*self.num\_classes

60 **for** feat **in** self.dataset.input\_features]

##### 10.1. K-means 179

61 **for** eg **in** self.dataset.train:

62 cl = self.class\_of\_eg(eg)

63 new\_class\_counts[cl] += 1

64 **for** (ind,feat) **in enumerate**(self.dataset.input\_features):

65 new\_feature\_sum[ind][cl] += feat(eg)

66 stable = (new\_class\_counts == self.class\_counts) **and** (self.feature\_sum == new\_feature\_sum)

67 self.class\_counts = new\_class\_counts

68 self.feature\_sum = new\_feature\_sum

69 self.num\_iterations += 1

70 **return** stable

71

72

73 **def** learn(self,n=100):

74 """do n steps of k-means, or until convergence"""

75 i=0

76 stable = False

77 **while** i<n **and not** stable:

78 stable = self.k\_means\_step()

79 i += 1

80 self.display(1,"Iteration",self.num\_iterations,

81 "class counts: ",self.class\_counts," Stable=",stable)

82 **return** stable

83

84 **def** show\_classes(self):

85 """sorts the data by the class and prints in order.

86 For visualizing small data sets

87 """

88 class\_examples = [[] **for** i **in range**(self.num\_classes)]

89 **for** eg **in** self.dataset.train:

90 class\_examples[self.class\_of\_eg(eg)].append(eg)

91 **print**("Class","Example",sep='\t')

92 **for** cl **in range**(self.num\_classes):

|  |  |  |
| --- | --- | --- |
| 93 |  | **for** eg **in** class\_examples[cl]: |
| 94 |  | **print**(cl,\*eg,sep='\t') |
| 95 |  |  |
| 96 | **def** | plot\_error(self, maxstep=20): |
| 97 |  | """Plots the sum-of-suares error as a function of the number of steps""" |
| 98 |  | plt.ion() |
| 99 |  | plt.xlabel("step") |
| 100 |  | plt.ylabel("Ave sum-of-squares error") |
| 101 |  | train\_errors = [] |
| 102 |  | **if** self.dataset.test: |
| 103 |  | test\_errors = [] |
| 104 |  | **for** i **in range**(maxstep): |
| 105  106  107  108  109  110 | self.learn(1)  train\_errors.append( **sum**(self.distance(self.class\_of\_eg(eg),eg)  **for** eg **in** self.dataset.train)  /**len**(self.dataset.train))  **if** self.dataset.test:  test\_errors.append( **sum**(self.distance(self.class\_of\_eg(eg),eg) | |

111 **for** eg **in** self.dataset.test)

112 /**len**(self.dataset.test))

113 plt.plot(**range**(1,maxstep+1),train\_errors,

114 label=**str**(self.num\_classes)+" classes. Training set")

115 **if** self.dataset.test:

116 plt.plot(**range**(1,maxstep+1),test\_errors,

117 label=**str**(self.num\_classes)+" classes. Test set")

118 plt.legend()

119 plt.draw()

120

121 %data = Data\_from\_file('data/emdata1.csv', num\_train=10, target\_index=2000) % trivial example

122 data = Data\_from\_file('data/emdata2.csv', num\_train=10, target\_index=2000)

123 %data = Data\_from\_file('data/emdata0.csv', num\_train=14, target\_index=2000) % example **from** textbook

124 kml = K\_means\_learner(data,2)

125 num\_iter=4

126 **print**("Class assignment after",num\_iter,"iterations:")

127 kml.learn(num\_iter); kml.show\_classes()

128

129 # Plot the error

130 # km2=K\_means\_learner(data,2); km2.plot\_error(20) # 2 classes

131 # km3=K\_means\_learner(data,3); km3.plot\_error(20) # 3 classes

132 # km13=K\_means\_learner(data,13); km13.plot\_error(20) # 13 classes

133

134 # data = Data\_from\_file('data/carbool.csv', target\_index=2000,boolean\_features=True)

135 # kml = K\_means\_learner(data,3)

136 # kml.learn(20); kml.show\_classes()

137 # km3=K\_means\_learner(data,3); km3.plot\_error(20) # 3 classes

138 # km3=K\_means\_learner(data,30); km3.plot\_error(20) # 30 classes

**Exercise 10.1** Change *boolean features* = *True* flag to allow for numerical features. K-means assumes the features are numerical, so we want to make non-numerical features into numerical features (using characteristic functions) but we probably don’t want to change numerical features into Boolean.

**Exercise 10.2** If there are many classes, some of the classes can become empty (e.g., try 100 classes with carbool.csv). Implement a way to put some examples into a class, if possible. Two ideas are:

1. Initialize the classes with actual examples, so that the classes will not start empty. (Do the classes become empty?)
2. 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.

## EM

In the following definition, a class, *c*, is a integer in range [0, *num classes*). *i* is an index of a feature, so *feat*[*i*] is the *i*th feature, and a feature is a function from tuples to values. *val* is a value of a feature.

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

* + - *class counts* is a list such that *class counts*[*c*] is the number of tuples with

#### *class* = *c*, where each tuple is weighted by its probability, i.e.,

*class counts*[*c*] = ∑

*t*:*class*(*t*)=*c*

*P*(*t*)

* + - *feature counts* is a list such that *feature counts*[*i*][*val*][*c*] is the weighted count of the number of tuples *t* with *feat*[*i*](*t*) = *val* and *class*(*t*) = *c*, each tuple is weighted by its probability, i.e.,

*feature counts*[*i*][*val*][*c*] = ∑

*t*:*feat*[*i*](*t*)=*val* and*class*(*t*)=*c*

*P*(*t*)

learnEM.py — EM Learning

11 **from** learnProblem **import** Data\_set, Learner, Data\_from\_file

12 **import** random

13 **import** math

14 **import** matplotlib.pyplot as plt

15

16 **class** EM\_learner(Learner):

17 **def** init (self,dataset, num\_classes):

18 self.dataset = dataset

19 self.num\_classes = num\_classes

20 self.class\_counts = None

21 self.feature\_counts = None

#### The function *em step* goes though the training examples, and updates these counts. The first time it is run, when there is no model, it uses random distri- butions.

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, 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 for a tuple, given the current statistics.

*P*(*c* | *tple*) ∝ *P*(*c*) ∗ ∏ *P*(*Xi*=*tple*(*i*) | *c*)

*i*

= *class counts*[*c*] ∗ ∏ *feature counts*[*i*][*feati*(*tple*)][*c*]

*len*(*self* .*dataset*) *i*

##### class counts[c]

*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 the original tuple in the current model

42 """

43 feats = self.dataset.input\_features

44 unnorm = [prod(feature\_counts[i][feat(tple)][c]

45 **for** (i,feat) **in enumerate**(feats))/(class\_counts[c]\*\*(**len**(feats)-1))

46 **for** c **in range**(self.num\_classes)]

47 thesum = **sum**(unnorm)

48 **return** [un/thesum **for** un **in** unnorm]

*learn* does *n* steps of EM:

learnEM.py — (continued)

50 **def** learn(self,n):

51 """do n steps of em"""

52 **for** i **in range**(n):

53 self.class\_counts,self.feature\_counts = self.em\_step(self.class\_counts,

54 self.feature\_counts)

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

learnEM.py — (continued)

56 **def** show\_class(self,c):

57 """sorts the data by the class and prints in order.

58 For visualizing small data sets

59 """

60 sorted\_data = **sorted**((self.prob(tpl,self.class\_counts,self.feature\_counts)[c],

61 ind, # preserve ordering for equal probabilities

62 tpl)

63 **for** (ind,tpl) **in enumerate**(self.dataset.train))

64 **for** cc,r,tpl **in** sorted\_data:

65 **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:

*P*(*tple*) = ∑ *P*(*c*) ∗ ∏ *P*(*Xi*=*tple*(*i*) | *c*)

*c i*

= ∑ *cc*[*c*] ∗ ∏ *fc*[*i*][*feati*(*tple*)][*c*]

*c len*(*self* .*dataset*) *i*

*cc*[*c*]

#### where *cc* is the class count and *fc* is feature count. *len*(*self* .*dataset*) can be dis- tributed out of the sum, and *cc*[*c*] can be taken out of the product:

1 1

= *len*(*self* .*dataset*) ∑ *cc*[*c*]#*feats*−1 ∗ ∏ *fc*[*i*][*feati*(*tple*)][*c*]

*c i*

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

learnEM.py — (continued)

67 **def** logloss(self,tple):

68 """returns the logloss of the prediction on tple, which is -log(P(tple))

69 based on the current class counts and feature counts

70 """

71 feats = self.dataset.input\_features

72 res = 0

73 cc = self.class\_counts

74 fc = self.feature\_counts

75 **for** c **in range**(self.num\_classes):

76 res += prod(fc[i][feat(tple)][c]

77 **for** (i,feat) **in enumerate**(feats))/(cc[c]\*\*(**len**(feats)-1))

78 **if** res>0:

79 **return** -math.log2(res/**len**(self.dataset.train))

80 **else**:

81 **return float**("inf") #infinity

82

83 **def** plot\_error(self, maxstep=20):

84 """Plots the logloss error as a function of the number of steps"""

85 plt.ion()

86 plt.xlabel("step")

87 plt.ylabel("Ave Logloss (bits)")

88 train\_errors = []

89 **if** self.dataset.test:

90 test\_errors = []

91 **for** i **in range**(maxstep):

92 self.learn(1)

93 train\_errors.append( **sum**(self.logloss(tple) **for** tple **in** self.dataset.train)

94 /**len**(self.dataset.train))

95 **if** self.dataset.test:

96 test\_errors.append( **sum**(self.logloss(tple) **for** tple **in** self.dataset.test)

97 /**len**(self.dataset.test))

98 plt.plot(**range**(1,maxstep+1),train\_errors,

|  |  |  |
| --- | --- | --- |
| 99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124 | label=**str**(self.num\_classes)+" classes. Training set")  **if** self.dataset.test: plt.plot(**range**(1,maxstep+1),test\_errors,  label=**str**(self.num\_classes)+" classes. Test set") plt.legend()  plt.draw()  **def** prod(L):  """returns the product of the elements of L""" res = 1  **for** e **in** L:  res \*= e  **return** res  **def** random\_dist(k):  """generate k random numbers that sum to 1""" res = [random.random() **for** i **in range**(k)]  s = **sum**(res)  **return** [v/s **for** v **in** res]  data = Data\_from\_file('data/emdata2.csv', num\_train=10, target\_index=2000) eml = EM\_learner(data,2)  num\_iter=2  **print**("Class assignment after",num\_iter,"iterations:") eml.learn(num\_iter); eml.show\_class(0) | |
| 125 | # | Plot the error |
| 126 | # | em2=EM\_learner(data,2); em2.plot\_error(40) # 2 classes |
| 127 | # | em3=EM\_learner(data,3); em3.plot\_error(40) # 3 classes |
| 128 | # | em13=EM\_learner(data,13); em13.plot\_error(40) # 13 classes |
| 129 |  |  |
| 130 | # | data = Data\_from\_file('data/carbool.csv', target\_index=2000,boolean\_features=False) |
| 131 | # | [f.frange for f in data.input\_features] |
| 132 | # | eml = EM\_learner(data,3) |
| 133 | # | eml.learn(20); eml.show\_class(0) |
| 134 | # | em3=EM\_learner(data,3); em3.plot\_error(60) # 3 classes |
| 135 | # | em3=EM\_learner(data,30); em3.plot\_error(60) # 30 classes |

**Exercise 10.3** For the EM data, where there are naturally 2 classes, 3 classes does better on the training set after a while than 2 classes, but worse on the test set. Explain why. Hint: look what the 3 classes are. Use ”em3.show class(i)” for each of the classes *i* ∈ [0, 3).

**Exercise 10.4** Write code to plot the logloss as a function of the number of classes (from 1 to say 15) for a fixed number of iterations. (From the experience with the existing code, think about how many iterations is appropriate.)

Chapter 11

Multiagent Systems

## Minimax

Here we consider two-player zero-sum games. Here a player only wins when another player loses. This can be modeled as where there is a single utility which one agent (the maximizing agent) is trying minimize and the other agent (the minimizing agent) is trying to minimize.

* + 1. Creating a two-player game

masProblem.py — A Multiagent Problem

11 **from** display **import** Displayable

12

13 **class** Node(Displayable):

14 """A node in a search tree. It has a

15 name a string

16 isMax is True if it is a maximizing node, otherwise it is minimizing node

17 children is the list of children

18 value is what it evaluates to if it is a leaf.

19 """

20 **def** init (self, name, isMax, value, children):

21 self.name = name

22 self.isMax = isMax

23 self.value = value

24 self.allchildren = children

25

26 **def** isLeaf(self):

27 """returns true of this is a leaf node"""

28 **return** self.allchildren **is** None

29

#### 185

30 **def** children(self):

31 """returns the list of all children."""

32 **return** self.allchildren

33

34 **def** evaluate(self):

35 """returns the evaluation for this node if it is a leaf"""

36 **return** self.value

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

masProblem.py — (continued)

38 fig10\_5 = Node("a",True,None, [

39 Node("b",False,None, [

40 Node("d",True,None, [

41 Node("h",False,None, [

42 Node("h1",True,7,None),

43 Node("h2",True,9,None)]),

44 Node("i",False,None, [

45 Node("i1",True,6,None),

46 Node("i2",True,888,None)])]),

47 Node("e",True,None, [

48 Node("j",False,None, [

49 Node("j1",True,11,None),

50 Node("j2",True,12,None)]),

51 Node("k",False,None, [

52 Node("k1",True,888,None),

53 Node("k2",True,888,None)])])]),

54 Node("c",False,None, [

55 Node("f",True,None, [

56 Node("l",False,None, [

57 Node("l1",True,5,None),

58 Node("l2",True,888,None)]),

59 Node("m",False,None, [

60 Node("m1",True,4,None),

61 Node("m2",True,888,None)])]),

62 Node("g",True,None, [

63 Node("n",False,None, [

64 Node("n1",True,888,None),

65 Node("n2",True,888,None)]),

66 Node("o",False,None, [

67 Node("o1",True,888,None),

68 Node("o2",True,888,None)])])])])

#### The following is a representation of a **magic-sum game**, where players take turns picking a number in the range [1, 9], and the first player to have 3 num- bers that sum to 15 wins. Note that this is a syntactic variant of **tic-tac-toe** or **naughts and crosses**. To see this, consider the numbers on a **magic square** (Fig- ure [11.1);](#_bookmark189) 3 numbers that add to 15 correspond exactly to the winning positions of tic-tac-toe played on the magic square.

Note that we do not remove symmetries. (What are the symmetries? How

|  |  |  |
| --- | --- | --- |
| 6 | 1 | 8 |
| 7 | 5 | 3 |
| 2 | 9 | 4 |

Figure 11.1: Magic Square

do the symmetries of tic-tac-toe translate here?)

masProblem.py — (continued)

70

71 **class** Magic\_sum(Node):

72 **def** init (self, xmove=True, last\_move=None,

73 available=[1,2,3,4,5,6,7,8,9], x=[], o=[]):

74 """This is a node in the search for the magic-sum game.

75 xmove is True if the next move belongs to X.

76 last\_move is the number selected in the last move

77 available is the list of numbers that are available to be chosen

78 x is the list of numbers already chosen by x

79 o is the list of numbers already chosen by o

80 """

81 self.isMax = self.xmove = xmove

82 self.last\_move = last\_move

83 self.available = available

84 self.x = x

85 self.o = o

86 self.allchildren = None #computed on demand

87 lm = **str**(last\_move)

88 self.name = "start" **if not** last\_move **else** "o="+lm **if** xmove **else** "x="+lm

89

90 **def** children(self):

91 **if** self.allchildren **is** None:

92 **if** self.xmove:

93 self.allchildren = [

94 Magic\_sum(xmove = **not** self.xmove,

95 last\_move = sel,

96 available = [e **for** e **in** self.available **if** e **is not** sel],

97 x = self.x+[sel],

98 o = self.o)

99 **for** sel **in** self.available]

100 **else**:

101 self.allchildren = [

102 Magic\_sum(xmove = **not** self.xmove,

103 last\_move = sel,

104 available = [e **for** e **in** self.available **if** e **is not** sel],

105 x = self.x,

106 o = self.o+[sel])

107 **for** sel **in** self.available]

108 **return** self.allchildren

109

110 **def** isLeaf(self):

111 """A leaf has no numbers available or is a win for one of the players.

112 We only need to check for a win for o if it is currently x's turn,

113 and only check for a win for x if it is o's turn (otherwise it would

114 have been a win earlier).

115 """

116 **return** (self.available == [] **or**

117 (sum\_to\_15(self.last\_move,self.o)

118 **if** self.xmove

119 **else** sum\_to\_15(self.last\_move,self.x)))

120

121 **def** evaluate(self):

122 **if** self.xmove **and** sum\_to\_15(self.last\_move,self.o):

123 **return** -1

124 **elif not** self.xmove **and** sum\_to\_15(self.last\_move,self.x):

125 **return** 1

126 **else**:

127 **return** 0

128

129 **def** sum\_to\_15(last,selected):

130 """is true if last, toegether with two other elements of selected sum to 15.

131 """

132 **return any**(last+a+b == 15

133 **for** a **in** selected **if** a != last

134 **for** b **in** selected **if** b != last **and** b != a)

11.1.2 Minimax and *α*-*β* Pruning

This is a naive depth-first **minimax algorithm**:

masMiniMax.py — Minimax search with alpha-beta pruning

11 **def** minimax(node):

12 """returns the value of node, and a best path for the agents

13 """

14 **if** node.isLeaf():

15 **return** node.evaluate(),None

16 **elif** node.isMax:

17 max\_score = -999

18 max\_path = None

19 **for** C **in** node.children():

20 score,path = minimax(C,depth+1)

21 **if** score > max\_score:

22 max\_score = score

23 max\_path = C.name,path

24 **return** max\_score,max\_path

25 **else**:

26 min\_score = 999

27 min\_path = None

28 **for** C **in** node.children():

29 score,path = minimax(C,depth+1)

30 **if** score < min\_score:

31 min\_score = score

32 min\_path = C.name,path

33 **return** min\_score,min\_path

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

masMiniMax.py — (continued)

35 **def** minimax\_alpha\_beta(node,alpha,beta,depth=0):

36 """node is a Node, alpha and beta are cutoffs, depth is the depth

37 returns value, path

38 where path is a sequence of nodes that results in the value"""

39 node.display(2," "\*depth,"minimax\_alpha\_beta(",node.name,", ",alpha, ", ", beta,")")

40 best=None # only used if it will be pruned

41 **if** node.isLeaf():

42 node.display(2," "\*depth,"returning leaf value",node.evaluate())

43 **return** node.evaluate(),None

44 **elif** node.isMax:

45 **for** C **in** node.children():

46 score,path = minimax\_alpha\_beta(C,alpha,beta,depth+1)

47 **if** score >= beta: # beta pruning

48 node.display(2," "\*depth,"pruned due to beta=",beta,"C=",C.name)

49 **return** score, None

50 **if** score > alpha:

51 alpha = score

52 best = C.name, path

53 node.display(2," "\*depth,"returning max alpha",alpha,"best",best)

54 **return** alpha,best

55 **else**:

56 **for** C **in** node.children():

57 score,path = minimax\_alpha\_beta(C,alpha,beta,depth+1)

58 **if** score <= alpha: # alpha pruning

59 node.display(2," "\*depth,"pruned due to alpha=",alpha,"C=",C.name)

60 **return** score, None

61 **if** score < beta:

62 beta=score

63 best = C.name,path

64 node.display(2," "\*depth,"returning min beta",beta,"best=",best)

65 **return** beta,best

#### Testing:

masMiniMax.py — (continued)

67 **from** masProblem **import** fig10\_5, Magic\_sum, Node

68

69 # Node.max\_display\_level=2 # print detailed trace

70 # minimax\_alpha\_beta(fig10\_5, -9999, 9999,0)

71 # minimax\_alpha\_beta(Magic\_sum(), -9999, 9999,0)

72

73 #To see how much time alpha-beta pruning can save over minimax, uncomment the following:

74 ## import timeit

75 ## timeit.Timer("minimax(Magic\_sum())",setup="from main import minimax, Magic\_sum"

76 ## ).timeit(number=1)

77 ## trace=False

78 ## timeit.Timer("minimax\_alpha\_beta(Magic\_sum(), -9999, 9999,0)",

79 ## setup="from main import minimax\_alpha\_beta, Magic\_sum"

80 ## ).timeit(number=1)

# Chapter 12

Reinforcement Learning

## Representing Agents and Environments

#### When the learning agent does an action in the environment, it observes a (*state*, *reward*)

pair from the environment. The *state* is the world state; this is the fully observ-

able assumption.

An RL environment implements a *do*(*action*) method that returns a (*state*, *reward*)

#### pair.

rlProblem.py — Representations for Reinforcement Learning

11 **import** random

12 **from** display **import** Displayable

13 **from** utilities **import** flip

14

15 **class** RL\_env(Displayable):

16 **def** init (self,actions,state):

17 self.actions = actions # set of actions

18 self.state = state # initial state

19

20 **def** do(self, action):

21 """do action

22 returns state,reward

23 """

24 **raise** NotImplementedError("RL\_env.do") # abstract method

#### Here is the definition of the simple 2-state, 2-action party/relax decision.

rlProblem.py — (continued)

26 **class** Healthy\_env(RL\_env):

27 **def** init (self):

28 RL\_env. init (self,["party","relax"], "healthy")

29

#### 191

30 **def** do(self, action):

31 """updates the state based on the agent doing action.

32 returns state,reward

33 """

34 **if** self.state=="healthy":

35 **if** action=="party":

36 self.state = "healthy" **if** flip(0.7) **else** "sick"

37 reward = 10

38 **else**: # action=="relax"

39 self.state = "healthy" **if** flip(0.95) **else** "sick"

40 reward = 7

41 **else**: # self.state=="sick"

42 **if** action=="party":

43 self.state = "healthy" **if** flip(0.1) **else** "sick"

44 reward = 2

45 **else**:

46 self.state = "healthy" **if** flip(0.5) **else** "sick"

47 reward = 0

48 **return** self.state,reward

* + 1. Simulating an environment from an MDP

#### Given the definition for an MDP (page [172),](#_bookmark174) *Env from MDP* takes in an MDP and simulates the environment with those dynamics.

Note that the MDP does not contain enough information to simulate a sys- tem, because it loses any dependency between the rewards and the resulting state; here we assume the agent always received the average reward for the state and action.

rlProblem.py — (continued)

50 **class** Env\_from\_MDP(RL\_env):

51 **def** init (self, mdp):

52 initial\_state = mdp.states[0]

53 RL\_env. init (self,mdp.actions, initial\_state)

54 self.mdp = mdp

55 self.action\_index = {action:index **for** (index,action) **in enumerate**(mdp.actions)}

56 self.state\_index = {state:index **for** (index,state) **in enumerate**(mdp.states)}

57

58 **def** do(self, action):

59 """updates the state based on the agent doing action.

60 returns state,reward

61 """

62 action\_ind = self.action\_index[action]

63 state\_ind = self.state\_index[self.state]

64 self.state = pick\_from\_dist(self.mdp.trans[state\_ind][action\_ind], self.mdp.states)

65 reward = self.mdp.reward[state\_ind][action\_ind]

66 **return** self.state, reward

67

68 **def** pick\_from\_dist(dist,values):

4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P1 | R |  |  | P2 |
|  |  | M |  |  |
|  |  |  |  | M |
| M | M |  | M |  |
| P3 |  |  |  | P4 |

3

2

1

0

0 1 2 3 4

#### Figure 12.1: Monster game

69 """

70 e.g. pick\_from\_dist([0.3,0.5,0.2],['a','b','c']) should pick 'a' with probability 0.3, etc.

71 """

72 ran = random.random()

73 i=0

74 **while** ran>dist[i]:

75 ran -= dist[i]

76 i += 1

77 **return** values[i]

* + 1. Simple Game

#### This is for the game depicted in Figure [12.1.](#_bookmark197)

rlSimpleEnv.py — Simple game

11 **import** random

12 **from** utilities **import** flip

13 **from** rlProblem **import** RL\_env

14

15 **class** Simple\_game\_env(RL\_env):

16 xdim = 5

17 ydim = 5

18

19 vwalls = [(0,3), (0,4), (1,4)] # vertical walls right of these locations

20 hwalls = [] # not implemented

21 crashed\_reward = -1

22

23 prize\_locs = [(0,0), (0,4), (4,0), (4,4)]

24 prize\_apears\_prob = 0.3

25 prize\_reward = 10

|  |  |  |
| --- | --- | --- |
| 26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | monster\_locs = [(0,1), (1,1), (2,3), (3,1), (4,2)]  monster\_appears\_prob = 0.4  monster\_reward\_when\_damaged = -10  repair\_stations = [(1,4)]  actions = ["up","down","left","right"]  **def** init (self): # State:  self.x = 2  self.y = 2 self.damaged = False self.prize = None  # Statistics self.number\_steps = 0  self.total\_reward = 0  self.min\_reward = 0  self.min\_step = 0  self.zero\_crossing = 0  RL\_env. init (self, Simple\_game\_env.actions,  (self.x, self.y, self.damaged, self.prize)) self.display(2,"","Step","Tot Rew","Ave Rew",sep="\t") | |
| 50 | **def** | do(self,action): |
| 51 |  | """updates the state based on the agent doing action. |
| 52 |  | returns state,reward |
| 53 |  | """ |
| 54 |  | reward = 0.0 |
| 55 |  | # A prize can appear: |
| 56 |  | **if** self.prize **is** None **and** flip(self.prize\_apears\_prob): |
| 57 |  | self.prize = random.choice(self.prize\_locs) |
| 58 |  | # Actions can be noisy |
| 59 |  | **if** flip(0.4): |
| 60 |  | actual\_direction = random.choice(self.actions) |
| 61 |  | **else**: |
| 62 |  | actual\_direction = action |
| 63 |  | # Modeling the actions given the actual direction |
| 64 |  | **if** actual\_direction == "right": |
| 65 |  | **if** self.x==self.xdim-1 **or** (self.x,self.y) **in** self.vwalls: |
| 66 |  | reward += self.crashed\_reward |
| 67 |  | **else**: |
| 68 |  | self.x += 1 |
| 69 |  | **elif** actual\_direction == "left": |
| 70 |  | **if** self.x==0 **or** (self.x-1,self.y) **in** self.vwalls: |
| 71 |  | reward += self.crashed\_reward |
| 72 |  | **else**: |
| 73 |  | self.x += -1 |
| 74 |  | **elif** actual\_direction == "up": |
| 75 |  | **if** self.y==self.ydim-1: |

76 reward += self.crashed\_reward

77 **else**:

78 self.y += 1

79 **elif** actual\_direction == "down":

80 **if** self.y==0:

81 reward += self.crashed\_reward

82 **else**:

83 self.y += -1

84 **else**:

85 **raise** RuntimeError("unknown\_direction "+**str**(direction))

86

87 # Monsters

88 **if** (self.x,self.y) **in** self.monster\_locs **and** flip(self.monster\_appears\_prob):

89 **if** self.damaged:

90 reward += self.monster\_reward\_when\_damaged

91 **else**:

92 self.damaged = True

93 **if** (self.x,self.y) **in** self.repair\_stations:

94 self.damaged = False

95

96 # Prizes

97 **if** (self.x,self.y) == self.prize:

98 reward += self.prize\_reward

99 self.prize = None

100

101 # Statistics

102 self.number\_steps += 1

103 self.total\_reward += reward

104 **if** self.total\_reward < self.min\_reward:

105 self.min\_reward = self.total\_reward

106 self.min\_step = self.number\_steps

107 **if** self.total\_reward>0 **and** reward>self.total\_reward:

108 self.zero\_crossing = self.number\_steps

109 self.display(2,"",self.number\_steps,self.total\_reward,

110 self.total\_reward/self.number\_steps,sep="\t")

111

112 **return** (self.x, self.y, self.damaged, self.prize), reward

### 12.1.3 Evaluation and Plotting

rlPlot.py — RL Plotter

11 **import** matplotlib.pyplot as plt

12

13 **def** plot\_rl(ag, label=None, yplot='Total', step\_size=None,

14 steps\_explore=1000, steps\_exploit=1000, xscale='linear'):

15 """

16 plots the agent ag

17 label is the label for the plot

18 yplot is 'Average' or 'Total'

19 step\_size is the number of steps between each point plotted

20 steps\_explore is the number of steps the agent spends exploring

21 steps\_exploit is the number of steps the agent spends exploiting

22 xscale is 'log' or 'linear'

23

24 returns total reward when exploring, total reward when exploiting

25 """

26 **assert** yplot **in** ['Average','Total']

27 **if** step\_size **is** None:

28 step\_size = **max**(1,(steps\_explore+steps\_exploit)//500)

29 **if** label **is** None:

30 label = ag.label

31 ag.max\_display\_level,old\_mdl = 1,ag.max\_display\_level

32 plt.ion()

33 plt.xscale(xscale)

34 plt.xlabel("step")

35 plt.ylabel(yplot+" reward")

36 steps = [] # steps

37 rewards = [] # return

38 ag.restart()

39 step = 0

40 **while** step < steps\_explore:

41 ag.do(step\_size)

42 step += step\_size

43 steps.append(step)

44 **if** yplot == "Average":

45 rewards.append(ag.acc\_rewards/step)

46 **else**:

47 rewards.append(ag.acc\_rewards)

48 acc\_rewards\_exploring = ag.acc\_rewards

49 ag.explore,explore\_save = 0,ag.explore

50 **while** step < steps\_explore+steps\_exploit:

51 ag.do(step\_size)

52 step += step\_size

53 steps.append(step)

54 **if** yplot == "Average":

55 rewards.append(ag.acc\_rewards/step)

56 **else**:

57 rewards.append(ag.acc\_rewards)

58 plt.plot(steps,rewards,label=label)

59 plt.legend(loc="upper left")

60 plt.draw()

61 ag.max\_display\_level = old\_mdl

62 ag.explore=explore\_save

63 **return** acc\_rewards\_exploring, ag.acc\_rewards-acc\_rewards\_exploring

## Q Learning

To run the Q-learning demo, in folder “aipython”, load “rlQTest.py”, and copy and paste the example queries at the bottom of that file. This assumes Python 3.

rlQLearner.py — Q Learning

11 **import** random

12 **from** display **import** Displayable

13 **from** utilities **import** argmax, flip

14

15 **class** RL\_agent(Displayable):

16 """An RL\_Agent

17 has percepts (s, r) for some state s and real reward r

18 """

rlQLearner.py — (continued)

20 **class** Q\_learner(RL\_agent):

21 """A Q-learning agent has

22 belief-state consisting of

23 state is the previous state

24 q is a {(state,action):value} dict

25 visits is a {(state,action):n} dict. n is how many times action was done in state

26 acc\_rewards is the accumulated reward

27

28 it observes (s, r) for some world-state s and real reward r

29 """

rlQLearner.py — (continued)

31 **def** init (self, env, discount, explore=0.1, fixed\_alpha=True, alpha=0.2,

32 alpha\_fun=**lambda** k:1/k,

33 qinit=0, label="Q\_learner"):

34 """env is the environment to interact with.

35 discount is the discount factor

36 explore is the proportion of time the agent will explore

37 fixed\_alpha specifies whether alpha is fixed or varies with the number of visits

38 alpha is the weight of new experiences compared to old experiences

39 alpha\_fun is a function that computes alpha from the number of visits

40 qinit is the initial value of the Q's

41 label is the label for plotting

42 """

43 RL\_agent. init (self)

44 self.env = env

45 self.actions = env.actions

46 self.discount = discount

47 self.explore = explore

48 self.fixed\_alpha = fixed\_alpha

49 self.alpha = alpha

50 self.alpha\_fun = alpha\_fun

51 self.qinit = qinit

52 self.label = label

53 self.restart()

#### restart is used to make the learner relearn everything. This is used by the plot- ter to create new plots.

rlQLearner.py — (continued)

55 **def** restart(self):

56 """make the agent relearn, and reset the accumulated rewards

57 """

58 self.acc\_rewards = 0

59 self.state = self.env.state

60 self.q = {}

61 self.visits = {}

#### *do* takes in the number of steps.

rlQLearner.py — (continued)

63 **def** do(self,num\_steps=100):

64 """do num\_steps of interaction with the environment"""

65 self.display(2,"s\ta\tr\ts'\tQ")

66 alpha = self.alpha

67 **for** i **in range**(num\_steps):

68 action = self.select\_action(self.state)

69 next\_state,reward = self.env.do(action)

70 **if not** self.fixed\_alpha:

71 k = self.visits[(self.state, action)] = self.visits.get((self.state, action),0)+1

72 alpha = self.alpha\_fun(k)

73 self.q[(self.state, action)] = (

74 (1-alpha) \* self.q.get((self.state, action),self.qinit)

75 + alpha \* (reward + self.discount

76 \* **max**(self.q.get((next\_state, next\_act),self.qinit)

77 **for** next\_act **in** self.actions)))

78 self.display(2,self.state, action, reward, next\_state,

79 self.q[(self.state, action)], sep='\t')

80 self.state = next\_state

81 self.acc\_rewards += reward

#### *select action* us used to select the next action to perform. This can be reimple- mented to give a different exploration strategy.

rlQLearner.py — (continued)

83 **def** select\_action(self, state):

84 """returns an action to carry out for the current agent

85 given the state, and the q-function

86 """

87 **if** flip(self.explore):

88 **return** random.choice(self.actions)

89 **else**:

90 **return** argmax((next\_act, self.q.get((state, next\_act),self.qinit))

91 **for** next\_act **in** self.actions)

**Exercise 12.1** Implement a soft-max action selection. Choose a temperature that works well for the domain. Explain how you picked this temperature. Compare the epsilon-greedy, soft-max and optimism in the face of uncertainty.

**Exercise 12.2** Implement SARSA. Hint: it does not do a *max* in *do*. Instead it needs to choose *next act* before it does the update.

* + 1. Testing Q-learning

#### The first tests are for the 2-action 2-state

rlQTest.py — RL Q Tester

11 **from** rlProblem **import** Healthy\_env

12 **from** rlQLearner **import** Q\_learner

13 **from** rlPlot **import** plot\_rl

14

15 env = Healthy\_env()

16 ag = Q\_learner(env, 0.7)

17 ag\_opt = Q\_learner(env, 0.7, qinit=100, label="optimistic" ) # optimistic agent

18 ag\_exp\_l = Q\_learner(env, 0.7, explore=0.01, label="less explore")

19 ag\_exp\_m = Q\_learner(env, 0.7, explore=0.5, label="more explore")

20 ag\_disc = Q\_learner(env, 0.9, qinit=100, label="disc 0.9")

21 ag\_va = Q\_learner(env, 0.7, qinit=100,fixed\_alpha=False,alpha\_fun=**lambda** k:10/(9+k),label="alpha=1

22

23 # ag.max\_display\_level = 2

24 # ag.do(20)

25 # ag.q # get the learned q-values

26 # ag.max\_display\_level = 1

27 # ag.do(1000)

28 # ag.q # get the learned q-values

29 # plot\_rl(ag,yplot="Average")

30 # plot\_rl(ag\_opt,yplot="Average")

31 # plot\_rl(ag\_exp\_l,yplot="Average")

32 # plot\_rl(ag\_exp\_m,yplot="Average")

33 # plot\_rl(ag\_disc,yplot="Average")

34 # plot\_rl(ag\_va,yplot="Average")

35

36 **from** mdpExamples **import** mdpt

37 **from** rlProblem **import** Env\_from\_MDP

38 envt = Env\_from\_MDP(mdpt)

39 agt = Q\_learner(envt, 0.8)

40 # agt.do(20)

41

42 **from** rlSimpleEnv **import** Simple\_game\_env

43 senv = Simple\_game\_env()

44 sag1 = Q\_learner(senv,0.9,explore=0.2,fixed\_alpha=True,alpha=0.1)

45 # plot\_rl(sag1,steps\_explore=100000,steps\_exploit=100000,label="alpha="+str(sag1.alpha))

46 sag2 = Q\_learner(senv,0.9,explore=0.2,fixed\_alpha=False)

47 # plot\_rl(sag2,steps\_explore=100000,steps\_exploit=100000,label="alpha=1/k")

48 sag3 = Q\_learner(senv,0.9,explore=0.2,fixed\_alpha=False,alpha\_fun=**lambda** k:10/(9+k))

49 # plot\_rl(sag3,steps\_explore=100000,steps\_exploit=100000,label="alpha=10/(9+k)")

## Model-based Reinforcement Learner

To run the demo, in folder “aipython”, load “rlModelLearner.py”, and copy and paste the example queries at the bottom of that file. This assumes Python 3.

#### A model-based reinforcement learner builds a Markov decision process model of the domain, simultaneously learns the model and plans with that model.

The model-based reinforcement learner used the following data structures:

* *q*[*s*, *a*] is dictionary that, given a (*s*, *a*) pair returns the *Q*-value, the esti- mate of the future (discounted) value of being in state *s* and doing action *a*.
* *r*[*s*, *a*] is dictionary that, given a (*s*, *a*) pair returns the average reward from doing *a* in state *s*.
* *t*[*s*, *a*, *s*j] is dictionary that, given a (*s*, *a*, *s*j) tuple returns the number of times *a* was done in state *s*, with the result being state *s*j.
* *visits*[*s*, *a*] is dictionary that, given a (*s*, *a*) pair returns the number of times action *a* was carried out in state *s*.
* *res states*[*s*, *a*] is dictionary that, given a (*s*, *a*) pair returns the list of re- sulting states that have occurred when action *a* was carried out in state *s*. This is used in the asynchronous value iteration to determine the *s*j states to sum over.
* *visits list* is a list of (*s*, *a*) pair that have been carried out. This is used to ensure there is no divide-by zero in the asynchronous value iteration. Note that this could be constructed from *r*, *visits* or *res states* by enumer- ating the keys, but needs to be a list for *random*.*choice*, and we don’t want to keep recreating it.

rlModelLearner.py — Model-based Reinforcement Learner

11 **import** random

12 **from** rlQLearner **import** RL\_agent

13 **from** display **import** Displayable

14 **from** utilities **import** argmax, flip

15

16 **class** Model\_based\_reinforcement\_learner(RL\_agent):

17 """A Model-based reinforcement learner

18 """

19

##### 12.3. Model-based Reinforcement Learner 201

20 **def** init (self, env, discount, explore=0.1, qinit=0,

21 updates\_per\_step=10, label="MBR\_learner"):

22 """env is the environment to interact with.

23 discount is the discount factor

24 explore is the proportion of time the agent will explore

25 qinit is the initial value of the Q's

26 updates\_per\_step is the number of AVI updates per action

27 label is the label for plotting

28 """

29 RL\_agent. init (self)

30 self.env = env

31 self.actions = env.actions

32 self.discount = discount

33 self.explore = explore

34 self.qinit = qinit

35 self.updates\_per\_step = updates\_per\_step

36 self.label = label

37 self.restart()

rlModelLearner.py — (continued)

39 **def** restart(self):

40 """make the agent relearn, and reset the accumulated rewards

41 """

42 self.acc\_rewards = 0

43 self.state = self.env.state

44 self.q = {} # {(st,action):q\_value} map

45 self.r = {} # {(st,action):reward} map

46 self.t = {} # {(st,action,st\_next):count} map

47 self.visits = {} # {(st,action):count} map

48 self.res\_states = {} # {(st,action):set\_of\_states} map

49 self.visits\_list = [] # list of (st,action)

50 self.previous\_action = None

rlModelLearner.py — (continued)

52 **def** do(self,num\_steps=100):

53 """do num\_steps of interaction with the environment

54 for each action, do updates\_per\_step iterations of asynchronous value iteration

55 """

56 **for** step **in range**(num\_steps):

57 pst = self.state # previous state

58 action = self.select\_action(pst)

59 self.state,reward = self.env.do(action)

60 self.acc\_rewards += reward

61 self.t[(pst,action,self.state)] = self.t.get((pst, action,self.state),0)+1

62 **if** (pst,action) **in** self.visits:

63 self.visits[(pst,action)] += 1

64 self.r[(pst,action)] += (reward-self.r[(pst,action)])/self.visits[(pst,action)]

65 self.res\_states[(pst,action)].add(self.state)

66 **else**:

67 self.visits[(pst,action)] = 1

68 self.r[(pst,action)] = reward

69 self.res\_states[(pst,action)] = {self.state}

70 self.visits\_list.append((pst,action))

71 st,act = pst,action #initial state-action pair for AVI

72 **for** update **in range**(self.updates\_per\_step):

73 self.q[(st,act)] = self.r[(st,act)]+self.discount\*(

74 **sum**(self.t[st,act,rst]/self.visits[st,act]\*

75 **max**(self.q.get((rst,nact),self.qinit) **for** nact **in** self.actions)

76 **for** rst **in** self.res\_states[(st,act)]))

77 st,act = random.choice(self.visits\_list)

rlModelLearner.py — (continued)

79 **def** select\_action(self, state):

80 """returns an action to carry out for the current agent

81 given the state, and the q-function

82 """

83 **if** flip(self.explore):

84 **return** random.choice(self.actions)

85 **else**:

86 **return** argmax((next\_act, self.q.get((state, next\_act),self.qinit))

87 **for** next\_act **in** self.actions)

rlModelLearner.py — (continued)

89 **from** rlQTest **import** senv # simple game environment

90 mbl1 = Model\_based\_reinforcement\_learner(senv,0.9,updates\_per\_step=10)

91 # plot\_rl(mbl1,steps\_explore=100000,steps\_exploit=100000,label="model-based(10)")

92 mbl2 = Model\_based\_reinforcement\_learner(senv,0.9,updates\_per\_step=1)

93 # plot\_rl(mbl2,steps\_explore=100000,steps\_exploit=100000,label="model-based(1)")

**Exercise 12.3** If there was only one update per step, the algorithm can be made simpler and use less space. Explain how. Does it make it more efficient? Is it worthwhile having more than one update per step for the games implemented here?

**Exercise 12.4** It is possible to implement the model-based reinforcement learner by replacing *q*, *r*, *visits*, *res states* with a single dictionary that returns a tuple (*q*, *r*, *v*, *tm*) where *q*, *r* and *v* are numbers, and *tm* is a map from resulting states into counts. Does this make the algorithm easier to understand? Does this make the algorithm more efficient?

**Exercise 12.5** If the states and the actions were mapped into integers, the dictio- naries could be implemented more efficiently as arrays. This entails an extra step in specifying problems. Implement this for the simple game. Is it more efficient?

## Reinforcement Learning with Features

To run the demo, in folder “aipython”, load “rlFeatures.py”, and copy and paste the example queries at the bottom of that file. This assumes Python 3.

* + 1. Representing Features

A feature is a function from state and action. To construct the features for a domain, we construct a function that takes a state and an action and returns the list of all feature values for that state and action. This feature set is redesigned for each problem.

*get features*(*state*, *action*) returns the feature values appropriate for the sim- ple game.

rlSimpleGameFeatures.py — Feature-based Reinforcement Learner

11 **from** rlSimpleEnv **import** Simple\_game\_env

12 **from** rlProblem **import** RL\_env

13

14 **def** get\_features(state,action):

15 """returns the list of feature values for the state-action pair

16 """

17 **assert** action **in** Simple\_game\_env.actions

18 (x,y,d,p) = state

19 # f1: would go to a monster

20 f1 = monster\_ahead(x,y,action)

21 # f2: would crash into wall

22 f2 = wall\_ahead(x,y,action)

23 # f3: action is towards a prize

24 f3 = towards\_prize(x,y,action,p)

25 # f4: damaged and action is toward repair station

26 f4 = towards\_repair(x,y,action) **if** d **else** 0

27 # f5: damaged and towards monster

28 f5 = 1 **if** d **and** f1 **else** 0

29 # f6: damaged

30 f6 = 1 **if** d **else** 0

31 # f7: not damaged

32 f7 = 1-f6

33 # f8: damaged and prize ahead

34 f8 = 1 **if** d **and** f3 **else** 0

35 # f9: not damaged and prize ahead

36 f9 = 1 **if not** d **and** f3 **else** 0

37 features = [1,f1,f2,f3,f4,f5,f6,f7,f8,f9]

38 **for** pr **in** Simple\_game\_env.prize\_locs+[None]:

39 **if** p==pr:

40 features += [x, 4-x, y, 4-y]

41 **else**:

42 features += [0, 0, 0, 0]

43 # fp04 feature for y when prize is at 0,4

44 # this knows about the wall to the right of the prize

45 **if** p==(0,4):

46 **if** x==0:

47 fp04 = y

48 **elif** y<3:

49 fp04 = y

50 **else**:

51

52 **else**:

fp04 = 4-y

53 fp04 = 0

54 features.append(fp04)

55 **return** features

56

57 **def** monster\_ahead(x,y,action):

58 """returns 1 if the location expected to get to by doing

59 action from (x,y) can contain a monster.

60 """

61 **if** action == "right" **and** (x+1,y) **in** Simple\_game\_env.monster\_locs:

62 **return** 1

63 **elif** action == "left" **and** (x-1,y) **in** Simple\_game\_env.monster\_locs:

64 **return** 1

65 **elif** action == "up" **and** (x,y+1) **in** Simple\_game\_env.monster\_locs:

66 **return** 1

67 **elif** action == "down" **and** (x,y-1) **in** Simple\_game\_env.monster\_locs:

68 **return** 1

69 **else**:

70 **return** 0

71

72 **def** wall\_ahead(x,y,action):

73 """returns 1 if there is a wall in the direction of action from (x,y).

74 This is complicated by the internal walls.

75 """

76 **if** action == "right" **and** (x==Simple\_game\_env.xdim-1 **or** (x,y) **in** Simple\_game\_env.vwalls):

77 **return** 1

78 **elif** action == "left" **and** (x==0 **or** (x-1,y) **in** Simple\_game\_env.vwalls):

79 **return** 1

80 **elif** action == "up" **and** y==Simple\_game\_env.ydim-1:

81 **return** 1

82 **elif** action == "down" **and** y==0:

83 **return** 1

84 **else**:

85 **return** 0

86

87 **def** towards\_prize(x,y,action,p):

88 """action goes in the direction of the prize from (x,y)"""

89 **if** p **is** None:

90 **return** 0

91 **elif** p==(0,4): # take into account the wall near the top-left prize

92 **if** action == "left" **and** (x>1 **or** x==1 **and** y<3):

93 **return** 1

94 **elif** action == "down" **and** (x>0 **and** y>2):

95 **return** 1

96 **elif** action == "up" **and** (x==0 **or** y<2):

97 **return** 1

98 **else**:

99 **return** 0

100 **else**:

101 px,py = p

102 **if** p==(4,4) **and** x==0:

103 **if** (action=="right" **and** y<3) **or** (action=="down" **and** y>2) **or** (action=="up" **and** y<2):

104 **return** 1

105 **else**:

106 **return** 0

107 **if** (action == "up" **and** y<py) **or** (action == "down" **and** py<y):

108 **return** 1

109 **elif** (action == "left" **and** px<x) **or** (action == "right" **and** x<px):

110 **return** 1

111 **else**:

112 **return** 0

113

114 **def** towards\_repair(x,y,action):

115 """returns 1 if action is towards the repair station.

116 """

117 **if** action == "up" **and** (x>0 **and** y<4 **or** x==0 **and** y<2):

118 **return** 1

119 **elif** action == "left" **and** x>1:

120 **return** 1

121 **elif** action == "right" **and** x==0 **and** y<3:

122 **return** 1

123 **elif** action == "down" **and** x==0 **and** y>2:

124 **return** 1

125 **else**:

126 **return** 0

127

128 **def** simp\_features(state,action):

129 """returns a list of feature values for the state-action pair

130 """

131 **assert** action **in** Simple\_game\_env.actions

132 (x,y,d,p) = state

133 # f1: would go to a monster

134 f1 = monster\_ahead(x,y,action)

135 # f2: would crash into wall

136 f2 = wall\_ahead(x,y,action)

137 # f3: action is towards a prize

138 f3 = towards\_prize(x,y,action,p)

139 **return** [1,f1,f2,f3]

12.4.2 Feature-based RL learner

#### This learns a linear function approximation of the Q-values. It requires the function *get features* that given a state and an action returns a list of values for all of the features. Each environment requires this function to be provided.

rlFeatures.py — Feature-based Reinforcement Learner

11 **import** random

12 **from** rlQLearner **import** RL\_agent

13 **from** display **import** Displayable

14 **from** utilities **import** argmax, flip

15

16 **class** SARSA\_LFA\_learner(RL\_agent):

17 """A SARSA\_LFA learning agent has

18 belief-state consisting of

19 state is the previous state

20 q is a {(state,action):value} dict

21 visits is a {(state,action):n} dict. n is how many times action was done in state

22 acc\_rewards is the accumulated reward

23

24 it observes (s, r) for some world-state s and real reward r

25 """

26 **def** init (self, env, get\_features, discount, explore=0.2, step\_size=0.01,

27 winit=0, label="SARSA\_LFA"):

28 """env is the feature environment to interact with

29 get\_features is a function get\_features(state,action) that returns the list of feature values

30 discount is the discount factor

31 explore is the proportion of time the agent will explore

32 step\_size is gradient descent step size

33 winit is the initial value of the weights

34 label is the label for plotting

35 """

36 RL\_agent. init (self)

37 self.env = env

38 self.get\_features = get\_features

39 self.actions = env.actions

40 self.discount = discount

41 self.explore = explore

42 self.step\_size = step\_size

43 self.winit = winit

44 self.label = label

45 self.restart()

#### *restart*() is used to make the learner relearn everything. This is used by the plotter to create new plots.

rlFeatures.py — (continued)

47 **def** restart(self):

48 """make the agent relearn, and reset the accumulated rewards

49 """

50 self.acc\_rewards = 0

51 self.state = self.env.state

52 self.features = self.get\_features(self.state, **list**(self.env.actions)[0])

53 self.weights = [self.winit **for** f **in** self.features]

54 self.action = self.select\_action(self.state)

#### *do* takes in the number of steps.

rlFeatures.py — (continued)

56 **def** do(self,num\_steps=100):

57 """do num\_steps of interaction with the environment"""

58 self.display(2,"s\ta\tr\ts'\tQ\tdelta")

59 **for** i **in range**(num\_steps):

60 next\_state,reward = self.env.do(self.action)

61 self.acc\_rewards += reward

62 next\_action = self.select\_action(next\_state)

63 feature\_values = self.get\_features(self.state,self.action)

64 oldQ = dot\_product(self.weights, feature\_values)

65 nextQ = dot\_product(self.weights, self.get\_features(next\_state,next\_action))

66 delta = reward + self.discount \* nextQ - oldQ

67 **for** i **in range**(**len**(self.weights)):

68 self.weights[i] += self.step\_size \* delta \* feature\_values[i]

69 self.display(2,self.state, self.action, reward, next\_state,

70 dot\_product(self.weights, feature\_values), delta, sep='\t')

71 self.state = next\_state

72 self.action = next\_action

73

74 **def** select\_action(self, state):

75 """returns an action to carry out for the current agent

76 given the state, and the q-function.

77 This implements an epsilon-greedy approach

78 where self.explore is the probability of exploring.

79 """

80 **if** flip(self.explore):

81 **return** random.choice(self.actions)

82 **else**:

83 **return** argmax((next\_act, dot\_product(self.weights,

84 self.get\_features(state,next\_act)))

85 **for** next\_act **in** self.actions)

86

87 **def** show\_actions(self,state=None):

88 """prints the value for each action in a state.

89 This may be useful for debugging.

90 """

91 **if** state **is** None:

92 state = self.state

93 **for** next\_act **in** self.actions:

94 **print**(next\_act,dot\_product(self.weights, self.get\_features(state,next\_act)))

95

96 **def** dot\_product(l1,l2):

97 **return sum**(e1\*e2 **for** (e1,e2) **in zip**(l1,l2))

#### Test code:

rlFeatures.py — (continued)

100 **from** rlQTest **import** senv # simple game environment

101 **from** rlSimpleGameFeatures **import** get\_features, simp\_features

102 **from** rlPlot **import** plot\_rl

103

104 fa1 = SARSA\_LFA\_learner(senv, get\_features, 0.9, step\_size=0.01)

105 #fa1.max\_display\_level = 2

106 #fa1.do(20)

107 #plot\_rl(fa1,steps\_explore=10000,steps\_exploit=10000,label="SARSA\_LFA(0.01)")

108 fas1 = SARSA\_LFA\_learner(senv, simp\_features, 0.9, step\_size=0.01)

109 #plot\_rl(fas1,steps\_explore=10000,steps\_exploit=10000,label="SARSA\_LFA(simp)")

**Exercise 12.6** How does the step-size affect performance? Try different step sizes (e.g., 0.1, 0.001, other sizes in between). Explain the behaviour you observe. Which step size works best for this example. Explain what evidence you are basing your prediction on.

**Exercise 12.7** Does having extra features always help? Does it sometime help? Does whether it helps depend on the step size? Give evidence for your claims.

**Exercise 12.8** For each of the following first predict, then plot, then explain the behavour you observed:

1. SARSA LFA, Model-based learning (with 1 update per step) and Q-learning for 10,000 steps 20% exploring followed by 10,000 steps 100% exploiting
2. SARSA LFA, model-based learning and Q-learning for
   1. 100,000 steps 20% exploring followed by 100,000 steps 100% exploit
   2. 10,000 steps 20% exploring followed by 190,000 steps 100% exploit
3. Suppose your goal was to have the best accumulated reward after 200,000 steps. You are allowed to change the exploration rate at a fixed number of steps. For each of the methods, which is the best position to start exploiting more? Which method is better? What if you wanted to have the best reward after 10,000 or 1,000 steps?

Based on this evidence, explain when it is preferable to use SARSA LFA, Model- based learner, or Q-learning.

Important: you need to run each algorithm more than once. Your explanation should include the variability as well as the typical behavior.

## Learning to coordinate - UNFINISHED!!!!

Coordinating agents should implement the agent architecture. However, in that architecture, an agent calls the environment. That architecture was cho- sen because it was simple. However, it does not really work when there are multiple agents. In such cases, a coroutining architecture is more appropriate.

We assume there is an x-player, and a y-player. *game*[*xa*][*ya*][*ag*] gives value to the agent *ag* (ag=for the x-player) of the strategy of the x-agent doing *xa* and

the y-agent doing *ya*.

learnCoordinate.py — Learning to Coordinate

11 **from** learnProblem **import** Learner

12

13 soccer = [[(-0.6,0.6),(-0.3,0.3)],[(-0.2,0.2),(-0.9,0.9)]]]

14 football = [[(2,1),(0,0)],[(0,0),(1,2)]]

15 prisoners\_game = [[(100,100),(0,1100)],[(1100,0),(1000,1000)]]]

16

17 **class** Policy\_hill\_climbing(Learner):

18 **def** init (self,game)

# Chapter 13

Relational Learning

## 13.1 Collaborative Filtering

#### Based on gradient descent algorithm of Koren, Y., Bell, R. and Volinsky, C., Matrix Factorization Techniques for Recommender Systems, IEEE Computer 2009.

This assumes the form of the dataset from movielens ([http://grouplens.](http://grouplens.org/datasets/movielens/) [org/datasets/movielens/](http://grouplens.org/datasets/movielens/)). The rating are a set of (*user*, *item*, *rating*, *timestamp*) tuples.

relnCollFilt.py — Latent Property-based Collaborative Filtering

11 **import** random

12 **import** matplotlib.pyplot as plt

13 **import** urllib.request

14 **from** learnProblem **import** Learner

15 **from** display **import** Displayable

16

17 **class** CF\_learner(Learner):

18 **def** init (self,

19 rating\_set, # a Rating\_set object

20 rating\_subset = None, # subset of ratings to be used as training ratings

21 test\_subset = None, # subset of ratings to be used as test ratings

22 step\_size = 0.01, # gradient descent step size

23 reglz = 1.0, # the weight for the regularization terms

24 num\_properties = 10, # number of hidden properties

25 property\_range = 0.02 # properties are initialized to be between

26 # -property\_range and property\_range

27 ):

28 self.rating\_set = rating\_set

29 self.ratings = rating\_subset **or** rating\_set.training\_ratings # whichever is not empty

30 **if** test\_subset **is** None:

#### 209

31 self.test\_ratings = self.rating\_set.test\_ratings

32 **else**:

33 self.test\_ratings = test\_subset

34 self.step\_size = step\_size

35 self.reglz = reglz

36 self.num\_properties = num\_properties

37 self.num\_ratings = **len**(self.ratings)

38 self.ave\_rating = (**sum**(r **for** (u,i,r,t) **in** self.ratings)

39 /self.num\_ratings)

40 self.users = {u **for** (u,i,r,t) **in** self.ratings}

41 self.items = {i **for** (u,i,r,t) **in** self.ratings}

42 self.user\_bias = {u:0 **for** u **in** self.users}

43 self.item\_bias = {i:0 **for** i **in** self.items}

44 self.user\_prop = {u:[random.uniform(-property\_range,property\_range)

45 **for** p **in range**(num\_properties)]

46 **for** u **in** self.users}

47 self.item\_prop = {i:[random.uniform(-property\_range,property\_range)

48 **for** p **in range**(num\_properties)]

49 **for** i **in** self.items}

50 self.zeros = [0 **for** p **in range**(num\_properties)]

51 self.**iter**=0

52

53 **def** stats(self):

54 self.display(1,"ave sumsq error of mean for training=",

55 **sum**((self.ave\_rating-rating)\*\*2 **for** (user,item,rating,timestamp)

56 **in** self.ratings)/**len**(self.ratings))

57 self.display(1,"ave sumsq error of mean for test=",

58 **sum**((self.ave\_rating-rating)\*\*2 **for** (user,item,rating,timestamp)

59 **in** self.test\_ratings)/**len**(self.test\_ratings))

60 self.display(1,"error on training set",

61 self.evaluate(self.ratings))

62 self.display(1,"error on test set",

63 self.evaluate(self.test\_ratings))

*learn* carries out *num iter* steps of gradient descent.

relnCollFilt.py — (continued)

65 **def** prediction(self,user,item):

66 """Returns prediction for this user on this item.

67 The use of .get() is to handle users or items not in the training set.

68 """

69 **return** (self.ave\_rating

70 + self.user\_bias.get(user,0) #self.user\_bias[user]

71 + self.item\_bias.get(item,0) #self.item\_bias[item]

72 + **sum**([self.user\_prop.get(user,self.zeros)[p]\*self.item\_prop.get(item,self.zeros)[p]

73 **for** p **in range**(self.num\_properties)]))

74

75 **def** learn(self, num\_iter = 50):

76 """ do num\_iter iterations of gradient descent."""

77 **for** i **in range**(num\_iter):

78 self.**iter** += 1

79 abs\_error=0

80 sumsq\_error=0

81 **for** (user,item,rating,timestamp) **in** random.sample(self.ratings,**len**(self.ratings)):

82 error = self.prediction(user,item) - rating

83 abs\_error += **abs**(error)

84 sumsq\_error += error \* error

85 self.user\_bias[user] -= self.step\_size\*error

86 self.item\_bias[item] -= self.step\_size\*error

87 **for** p **in range**(self.num\_properties):

88 self.user\_prop[user][p] -= self.step\_size\*error\*self.item\_prop[item][p]

89 self.item\_prop[item][p] -= self.step\_size\*error\*self.user\_prop[user][p]

90 **for** user **in** self.users:

91 self.user\_bias[user] -= self.step\_size\*self.reglz\* self.user\_bias[user]

92 **for** p **in range**(self.num\_properties):

93 self.user\_prop[user][p] -= self.step\_size\*self.reglz\*self.user\_prop[user][p]

94 **for** item **in** self.items:

95 self.item\_bias[item] -= self.step\_size\*self.reglz\*self.item\_bias[item]

96 **for** p **in range**(self.num\_properties):

97 self.item\_prop[item][p] -= self.step\_size\*self.reglz\*self.item\_prop[item][p]

98 self.display(1,"Iteration",self.**iter**,

99 "(Ave Abs,AveSumSq) training =",self.evaluate(self.ratings),

100 "test =",self.evaluate(self.test\_ratings))

#### *evaluate* evaluates current predictions on the rating set:

relnCollFilt.py — (continued)

102 **def** evaluate(self,ratings):

103 """returns (avergage\_absolute\_error, average\_sum\_squares\_error) for ratings

104 """

105 abs\_error = 0

106 sumsq\_error = 0

107 **if not** ratings: **return** (0,0)

108 **for** (user,item,rating,timestamp) **in** ratings:

109 error = self.prediction(user,item) - rating

110 abs\_error += **abs**(error)

111 sumsq\_error += error \* error

112 **return** abs\_error/**len**(ratings), sumsq\_error/**len**(ratings)

* + 1. Alternative Formulation

#### An alternative formulation is to regularize after each update.

* + 1. Plotting

relnCollFilt.py — (continued)

114 **def** plot\_predictions(self, examples="test"):

115 """

116 examples is either "test" or "training" or the actual examples

117 """

118 **if** examples == "test":

119 examples = self.test\_ratings

120 **elif** examples == "training":

121 examples = self.ratings

122 plt.ion()

123 plt.xlabel("prediction")

124 plt.ylabel("cumulative proportion")

125 self.actuals = [[] **for** r **in range**(0,6)]

126 **for** (user,item,rating,timestamp) **in** examples:

127 self.actuals[rating].append(self.prediction(user,item))

128 **for** rating **in range**(1,6):

129 self.actuals[rating].sort()

130 numrat=**len**(self.actuals[rating])

131 yvals = [i/numrat **for** i **in range**(numrat)]

132 plt.plot(self.actuals[rating], yvals, label="rating="+**str**(rating))

133 plt.legend()

134 plt.draw()

#### This plots a single property. Each (*user*, *item*, *rating*) is plotted where the x-value is the value of the property for the user, the y-value is the value of the property for the item, and the rating is plotted at this (*x*, *y*) position. That is, *rating* is plotted at the (*x*, *y*) position (*p*(*user*), *p*(*item*)).

relnCollFilt.py — (continued)

136 **def** plot\_property(self,

137 p, # property

138 plot\_all=False, # true if all points should be plotted

139 num\_points=200 # number of random points plotted if not all

140 ):

141 """plot some of the user-movie ratings,

142 if plot\_all is true

143 num\_points is the number of points selected at random plotted.

144

145 the plot has the users on the x-axis sorted by their value on property p and

146 with the items on the y-axis sorted by their value on property p and

147 the ratings plotted at the corresponding x-y position.

148 """

149 plt.ion()

150 plt.xlabel("users")

151 plt.ylabel("items")

152 user\_vals = [self.user\_prop[u][p]

153 **for** u **in** self.users]

154 item\_vals = [self.item\_prop[i][p]

155 **for** i **in** self.items]

156 plt.axis([**min**(user\_vals)-0.02,

157 **max**(user\_vals)+0.05,

158 **min**(item\_vals)-0.02,

159 **max**(item\_vals)+0.05])

160 **if** plot\_all:

161 **for** (u,i,r,t) **in** self.ratings:

162 plt.text(self.user\_prop[u][p],

163

164

165

**else**:

self.item\_prop[i][p],

**str**(r))

166 **for** i **in range**(num\_points):

167 (u,i,r,t) = random.choice(self.ratings)

168 plt.text(self.user\_prop[u][p],

169 self.item\_prop[i][p],

170 **str**(r))

171 plt.show()

### 13.1.3 Creating Rating Sets

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

relnCollFilt.py — (continued)

173 **class** Rating\_set(Displayable):

174 **def** init (self,

175 date\_split=892000000,

176 local\_file=False,

177 [url="http://files.grouplens.org/datasets/movielens/ml-100k/u.data",](http://files.grouplens.org/datasets/movielens/ml-100k/u.data)

178 file\_name="u.data"):

179 self.display(1,"reading...")

180 **if** local\_file:

181 lines = **open**(file\_name,'r')

182 **else**:

183 lines = (line.decode('utf-8') **for** line **in** urllib.request.urlopen(url))

184 all\_ratings = (**tuple**(**int**(e) **for** e **in** line.strip().split('\t'))

185 **for** line **in** lines)

186 self.training\_ratings = []

187 self.training\_stats = {1:0, 2:0, 3:0, 4:0 ,5:0}

188 self.test\_ratings = []

189 self.test\_stats = {1:0, 2:0, 3:0, 4:0 ,5:0}

190 **for** rate **in** all\_ratings:

191 **if** rate[3] < date\_split: # rate[3] is timestamp

192 self.training\_ratings.append(rate)

193 self.training\_stats[rate[2]] += 1

194 **else**:

195 self.test\_ratings.append(rate)

196 self.test\_stats[rate[2]] += 1

197 self.display(1,"...read:", **len**(self.training\_ratings),"training ratings and",

198 **len**(self.test\_ratings),"test ratings")

199 tr\_users = {user **for** (user,item,rating,timestamp) **in** self.training\_ratings}

200 test\_users = {user **for** (user,item,rating,timestamp) **in** self.test\_ratings}

201 self.display(1,"users:",**len**(tr\_users),"training,",**len**(test\_users),"test,",

202 **len**(tr\_users & test\_users),"in common")

203 tr\_items = {item **for** (user,item,rating,timestamp) **in** self.training\_ratings}

204 test\_items = {item **for** (user,item,rating,timestamp) **in** self.test\_ratings}

205 self.display(1,"items:",**len**(tr\_items),"training,",**len**(test\_items),"test,",

206 **len**(tr\_items & test\_items),"in common")

207 self.display(1,"Rating statistics for training set: ",self.training\_stats)

208 self.display(1,"Rating statistics for test set: ",self.test\_stats)

#### Sometimes it is useful to plot a property for all (*user*, *item*, *rating*) triples. There are too many such triples in the data set. The method *create top subset* creates a much smaller dataset where this makes sense. It picks the most rated items, then picks the users who have the most ratings on these items. It is designed for depicting the meaning of properties, and may not be useful for other purposes.

relnCollFilt.py — (continued)

210 **def** create\_top\_subset(self, num\_items = 30, num\_users = 30):

211 """Returns a subset of the ratings by picking the most rated items,

212 and then the users that have most ratings on these, and then all of the

213 ratings that involve these users and items.

214 """

215 items = {item **for** (user,item,rating,timestamp) **in** self.training\_ratings}

216

217 item\_counts = {i:0 **for** i **in** items}

218 **for** (user,item,rating,timestamp) **in** self.training\_ratings:

219 item\_counts[item] += 1

220

221 items\_sorted = **sorted**((item\_counts[i],i) **for** i **in** items)

222 top\_items = items\_sorted[-num\_items:]

223 set\_top\_items = **set**(item **for** (count, item) **in** top\_items)

224

225 users = {user **for** (user,item,rating,timestamp) **in** self.training\_ratings}

226 user\_counts = {u:0 **for** u **in** users}

227 **for** (user,item,rating,timestamp) **in** self.training\_ratings:

228 **if** item **in** set\_top\_items:

229 user\_counts[user] += 1

230

231 users\_sorted = **sorted**((user\_counts[u],u)

232 **for** u **in** users)

233 top\_users = users\_sorted[-num\_users:]

234 set\_top\_users = **set**(user **for** (count, user) **in** top\_users)

235 used\_ratings = [ (user,item,rating,timestamp)

236 **for** (user,item,rating,timestamp) **in** self.training\_ratings

237 **if** user **in** set\_top\_users **and** item **in** set\_top\_items]

238 **return** used\_ratings

239

240 movielens = Rating\_set()

241 learner0 = CF\_learner(movielens, num\_properties = 1)

242 #learner0.learn(50)

243 # learner0.plot\_predictions(examples = "training")

244 # learner0.plot\_predictions(examples = "test")

245 #learner0.plot\_property(0)

246 #movielens\_subset = movielens.create\_top\_subset(num\_items = 20, num\_users = 20)

247 #learner1 = CF\_learner(movielens, rating\_subset=movielens\_subset, test\_subset=[], num\_properties=1)

248 #learner1.learn(1000)

249 #learner1.plot\_property(0,plot\_all=True)