Python code for Artificial Intelligence: Foundations of Computational Agents (CPSC 322 Edition)

David L. Poole and Alan K. Mackworth

Version 0.8.2 of September 28, 2020.

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

Contents

Chapter 1

Python for Artificial Intelligence

1.1 Why Python?

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

Python is reasonably efficient. Efficiency is usually not a problem for small examples. If your Python code is not efficient enough, a general procedure to improve it is to find out what is taking most 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.

1.2 Getting Python

You need Python 3 (`http://python.org/`) and matplotlib (`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()`.

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://ipython.org/`). To install ipython after you have installed python do:

```
pip3 install ipython
```

1.3 Running Python

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

Here we describe the most simple version that uses no IDE. If you download the zip file, and cd to the "`aipython_322`" 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 important to enter interactive mode), with user input in bold:

```
$ ipython -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.
```

```
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 download 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 exercises can be done (and should be done) without using other libraries; the aim is for you to spend your time thinking about how to solve the problem rather than searching for pre-existing solutions.

1.4 Pitfalls

It is important to know when side effects occur. Often AI programs consider what would 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 appropriate.

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

1.5 Features of Python

1.5.1 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 element (advancing the iterator) and raises a StopIteration exception if there is no next element. The following shows a simple example, where user input is prepended with >>>

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

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

Comprehensions can also be used for dictionaries. The following code creates an index for list *a*:

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

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

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

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

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

1.5.2 Functions as first-class objects

Python can create lists and other data structures that contain functions. There is an issue that tricks many newcomers to Python. For a local variable in a function, the function uses the last value of the variable when the function is

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

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

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).

[1] Numbered lines are Python code available in the code-directory, `aipython_322`. 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 definitions.

1.5.3 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 argument, as the single argument acts as the second argument of the function. Note also that the built-in range also allows for indexing (e.g., $range(2,30,3)[2]$ returns 8), which the above implementation does not. However *myrange* also works for floats, which the built-in range does not.

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

Yield can be used to generate the same sequence of values as in the example of Section 1.5.1:

pythonDemo.py — (continued)

```
49 def ga(n):
50     """generates square of even nonnegative integers less than n"""
51     for e in range(n):
52         if e%2==0:
53             yield e*e
54 a = ga(20)
```

The sequence of next(a), and list(a) gives exactly the same results as the comprehension in Section 1.5.1.

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.

1.6 Useful Libraries

1.6.1 Timing Code

In order to compare algorithms, we often want to compute how long a program takes; this is called the **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")
```

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

```
start_time = time.perf_counter(); compute_for_a_while(); end_time = time.perf_counter()
```

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

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

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

1.6.2 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)^2/10-20"""
79     return (x-40)**2/10-20
80
81 # Try the following:
82 # from pythonDemo import myplot, slin, sqfun
83 # import matplotlib.pyplot as plt
84 # myplot(0,100,1,slin,sqfun)
85 # plt.legend(loc="best")
86 # import math
87 # plt.plot([41+40*math.cos(th/10) for th in range(50)],
88 #          [100+100*math.sin(th/10) for th in range(50)])
89 # plt.text(40,100,"ellipse?")
90 # plt.xscale('log')
```

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

1.7 Utilities

1.7.1 Display

In this distribution, to keep things simple and to only use standard Python, we use a text-oriented tracing of the code. A graphical depiction of the code could

override the definition of *display* (but we leave it as a project).

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

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

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

The definition of *display* is:

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

```
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 dictionary of the keyword arguments). This will not work in Python 2, and will give an error.

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

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

$$Classname.max\_display\_level = 3$$

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

The value of *max_display_level* by convention is:

0 display nothing

1 display solutions (nothing that happens repeatedly)

2 also display the values as they change (little detail through a loop)

3 also display more details

4 and above even more detail

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

display.py — (continued)

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

1.7.2 Argmax

Python has a built-in *max* function that takes a generator (or a list or set) and returns the maximum value. The *argmax* method returns the index of an element that has the maximum value. If there are multiple elements with the maximum value, one 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.

1.7.3 Probability

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

utilities.py — (continued)

```
30 def flip(prob):
```

```
31      """return true with probability prob"""
32      return random.random() < prob
```

1.7.4 Dictionary Union

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

utilities.py — (continued)

```
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
```

1.8 Testing Code

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

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

In your code you should do more substantial testing than 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

Searching for Solutions

2.1 Representing Search Problems

A search problem consists of:

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

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

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

searchProblem.py — representations of search problems

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

```
17      The methods must be overridden to define a search problem."""
18
19      def start_node(self):
20          """returns start node"""
21          raise NotImplementedError("start_node") # abstract method
22
23      def is_goal(self,node):
24          """is True if node is a goal"""
25          raise NotImplementedError("is_goal") # abstract method
26
27      def neighbors(self,node):
28          """returns a list 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 $\langle$*from_node*, *to_node*$\rangle$, but can also contain a non-negative *cost* (which defaults to 1) and can be labeled with an *action*.

searchProblem.py — (continued)

```
36  class Arc(object):
37      """An arc has a from_node and a to_node node and a (non-negative) 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)
```

2.1.1 Explicit Representation of Search Graph

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

An **explicit graph** consists of

- a list or set of nodes
- a list or set of arcs

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

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

searchProblem.py — (continued)

```
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     * a dictionary that maps each node into its (x,y) position
61     """
62 
63     def __init__(self, nodes, arcs, start=None, goals=set(), hmap={}, positions={}):
64         self.neighs = {}
65         self.nodes = nodes
66         for node in nodes:
67             self.neighs[node]=[]
68         self.arcs = arcs
69         for arc in arcs:
70             self.neighs[arc.from_node].append(arc)
71         self.start = start
72         self.goals = goals
73         self.hmap = hmap
74         self.positions = positions
75 
76     def start_node(self):
77         """returns start node"""
78         return self.start
79 
80     def is_goal(self,node):
81         """is True if node is a goal"""
82         return node in self.goals
83 
84     def neighbors(self,node):
85         """returns the neighbors of node"""
86         return self.neighs[node]
87 
88     def heuristic(self,node):
89         """Gives the heuristic value of node n.
90         Returns 0 if not overridden in the hmap."""
91         if node in self.hmap:
92             return self.hmap[node]
```

```
93          else:
94              return 0
95
96      def __repr__(self):
97          """returns a string representation of the search problem"""
98          res=""
99          for arc in self.arcs:
100             res += str(arc)+". "
101         return res
```

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

searchProblem.py — (continued)

```
103     def neighbor_nodes(self,node):
104         """returns an iterator over the neighbors of node"""
105         return (path.to_node for path in self.neighs[node])
```

2.1.2 Paths

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

A path is either:

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

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

searchProblem.py — (continued)

```
107 class Path(object):
108     """A path is either a node or a path followed by an arc"""
109
110     def __init__(self,initial,arc=None):
111         """initial is either a node (in which case arc is None) or
112         a path (in which case arc is an object of type Arc)"""
113         self.initial = initial
114         self.arc=arc
115         if arc is None:
116             self.cost=0
117         else:
118             self.cost = initial.cost+arc.cost
119
120     def end(self):
```

```
        """returns the node at the end of the path"""
        if self.arc is None:
            return self.initial
        else:
            return self.arc.to_node

    def nodes(self):
        """enumerates the nodes for the path.
        This starts at the end and enumerates nodes in the path backwards."""
        current = self
        while current.arc is not None:
            yield current.arc.to_node
            current = current.initial
        yield current.initial

    def initial_nodes(self):
        """enumerates the nodes for the path before the end node.
        This starts at the end and enumerates nodes in the path backwards."""
        if self.arc is not None:
            for nd in self.initial.nodes(): yield nd # could be "yield from"

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

2.1.3 Example Search Problems

The first search problem is one with 5 nodes where the least-cost path is one with many arcs. See Figure 2.1. Note that this example is used for the unit tests, so the test (in `searchGeneric`) will need to be changed if this is changed.

searchProblem.py — (continued)

```
problem1 = Search_problem_from_explicit_graph(
    {'a','b','c','d','g'},
    [Arc('a','c',1), Arc('a','b',3), Arc('c','d',3), Arc('c','b',1),
        Arc('b','d',1), Arc('b','g',3), Arc('d','g',1)],
    start = 'a',
    goals = {'g'},
    positions={'a': (0, 0), 'b': (1, 1), 'c': (0,1), 'd': (1,2), 'g': (2,2)})
```

The second search problem is one with 8 nodes where many paths do not lead to the goal. See Figure 2.2.

searchProblem.py — (continued)

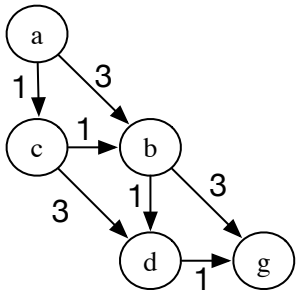


Figure 2.1: problem1

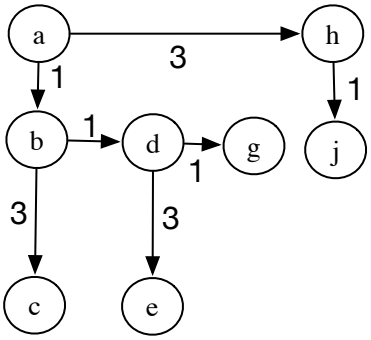


Figure 2.2: problem2

```
problem2 = Search_problem_from_explicit_graph(
    {'a','b','c','d','e','g','h','j'},
    [Arc('a','b',1), Arc('b','c',3), Arc('b','d',1), Arc('d','e',3),
        Arc('d','g',1), Arc('a','h',3), Arc('h','j',1)],
    start = 'a',
    goals = {'g'},
    positions={'a': (0, 0), 'b': (0, 1), 'c': (0,4), 'd': (1,1), 'e': (1,4),
                  'g': (2,1), 'h': (3,0), 'j': (3,1)})
```

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

searchProblem.py — (continued)

```
problem3 = Search_problem_from_explicit_graph(
    {'a','b','c','d','e','g','h','j'},
    [],
    start = 'g',
    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)

```
acyclic_delivery_problem = Search_problem_from_explicit_graph(
   {'mail','ts','o103','o109','o111','b1','b2','b3','b4','c1','c2','c3',
    'o125','o123','o119','r123','storage'},
    [Arc('ts','mail',6),
       Arc('o103','ts',8),
       Arc('o103','b3',4),
       Arc('o103','o109',12),
       Arc('o109','o119',16),
       Arc('o109','o111',4),
       Arc('b1','c2',3),
       Arc('b1','b2',6),
       Arc('b2','b4',3),
       Arc('b3','b1',4),
       Arc('b3','b4',7),
       Arc('b4','o109',7),
       Arc('c1','c3',8),
       Arc('c2','c3',6),
       Arc('c2','c1',4),
       Arc('o123','o125',4),
       Arc('o123','r123',4),
       Arc('o119','o123',9),
       Arc('o119','storage',7)],
   start = 'o103',
   goals = {'r123'},
   hmap = {
       'mail' : 26,
       'ts' : 23,
       'o103' : 21,
       'o109' : 24,
       'o111' : 27,
       'o119' : 11,
       'o123' : 4,
       'o125' : 6,
       'r123' : 0,
       'b1' : 13,
       'b2' : 15,
       'b3' : 17,
       'b4' : 18,
       'c1' : 6,
       'c2' : 10,
       'c3' : 12,
       'storage' : 12
       }
   )
```

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)

```
cyclic_delivery_problem = Search_problem_from_explicit_graph(
   {'mail','ts','o103','o109','o111','b1','b2','b3','b4','c1','c2','c3',
    'o125','o123','o119','r123','storage'},
    [  Arc('ts','mail',6), Arc('mail','ts',6),
       Arc('o103','ts',8), Arc('ts','o103',8),
       Arc('o103','b3',4),
       Arc('o103','o109',12), Arc('o109','o103',12),
       Arc('o109','o119',16), Arc('o119','o109',16),
       Arc('o109','o111',4), Arc('o111','o109',4),
       Arc('b1','c2',3),
       Arc('b1','b2',6), Arc('b2','b1',6),
       Arc('b2','b4',3), Arc('b4','b2',3),
       Arc('b3','b1',4), Arc('b1','b3',4),
       Arc('b3','b4',7), Arc('b4','b3',7),
       Arc('b4','o109',7),
       Arc('c1','c3',8), Arc('c3','c1',8),
       Arc('c2','c3',6), Arc('c3','c2',6),
       Arc('c2','c1',4), Arc('c1','c2',4),
       Arc('o123','o125',4), Arc('o125','o123',4),
       Arc('o123','r123',4), Arc('r123','o123',4),
       Arc('o119','o123',9), Arc('o123','o119',9),
       Arc('o119','storage',7), Arc('storage','o119',7)],
   start = 'o103',
   goals = {'r123'},
   hmap = {
       'mail' : 26,
       'ts' : 23,
       'o103' : 21,
       'o109' : 24,
       'o111' : 27,
       'o119' : 11,
       'o123' : 4,
       'o125' : 6,
       'r123' : 0,
       'b1' : 13,
       'b2' : 15,
       'b3' : 17,
       'b4' : 18,
       'c1' : 6,
       'c2' : 10,
       'c3' : 12,
       'storage' : 12
       }
   )
```

2.2 Generic Searcher and Variants

To run the search demos, in folder "aipython_322", 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.

2.2.1 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*

```
from display import Displayable, visualize

class Searcher(Displayable):
    """returns a searcher for a problem.
    Paths can be found by repeatedly calling search().
    This does depth-first search unless overridden
    """
    def __init__(self, problem):
        """creates a searcher from a problem
        """
        self.problem = problem
        self.initialize_frontier()
        self.num_expanded = 0
        self.add_to_frontier(Path(problem.start_node()))
        super().__init__()

    def initialize_frontier(self):
        self.frontier = []

    def empty_frontier(self):
        return self.frontier == []

    def add_to_frontier(self,path):
        self.frontier.append(path)

    @visualize
    def search(self):
        """returns (next) path from the problem's start node
        to a goal node.
        Returns None if no path exists.
        """
        while not self.empty_frontier():
            path = self.frontier.pop()
            self.display(2, "Expanding:",path,"(cost:",path.cost,")")
            self.num_expanded += 1
```

```
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(list(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), and *list* is needed if the neighboure are generated. Reversing the neighbours might not be required for other methods. The calls to *reversed* and *list* can be removed, and the algothihm still implements depth-fist search.

Exercise 2.1 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.

2.2.2 Frontier as a Priority Queue

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

Following the lead of the Python documentation, `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):
```

```
    """A frontier consists of a priority queue (heap), frontierpq, of
        (value, index, path) triples, where
    * value is the value we want to minimize (e.g., path cost + h).
    * index is a unique index for each element
    * path is the path on the queue
    Note that the priority queue always returns the smallest element.
    """

    def __init__(self):
        """constructs the frontier, initially an empty priority queue
        """
        self.frontier_index = 0 # the number of items ever added to the frontier
        self.frontierpq = [] # the frontier priority queue

    def empty(self):
        """is True if the priority queue is empty"""
        return self.frontierpq == []

    def add(self, path, value):
        """add a path to the priority queue
        value is the value to be minimized"""
        self.frontier_index += 1 # get a new unique index
        heapq.heappush(self.frontierpq,(value, -self.frontier_index, path))

    def pop(self):
        """returns and removes the path of the frontier with minimum value.
        """
        (_,_,path) = heapq.heappop(self.frontierpq)
        return path
```

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

searchGeneric.py — (continued)

```
    def count(self,val):
        """returns the number of elements of the frontier with value=val"""
        return sum(1 for e in self.frontierpq if e[0]==val)

    def __repr__(self):
        """string representation of the frontier"""
        return str([(n,c,str(p)) for (n,c,p) in self.frontierpq])

    def __len__(self):
        """length of the frontier"""
        return len(self.frontierpq)

    def __iter__(self):
        """iterate through the paths in the frontier"""
        for (_,_,path) in self.frontierpq:
            yield path
```

2.2.3 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)
```

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

searchGeneric.py — (continued)

```
130 import searchProblem as searchProblem
131
132 def test(SearchClass, problem=searchProblem.problem1, solutions=[['g','d','b','c','a']] ):
133     """Unit test for aipython searching algorithms.
134     SearchClass is a class that takes a problemm and implements search()
135     problem is a search problem
136     solutions is a list of optimal solutions
137     """
138     print("Testing problem 1:")
139     schr1 = SearchClass(problem)
140     path1 = schr1.search()
141     print("Path found:",path1)
142     assert path1 is not None, "No path is found in problem1"
143     assert list(path1.nodes()) in solutions, "Shortest path not found in problem1"
144     print("Passed unit test")
145
146 if __name__ == "__main__":
147     #test(Searcher)
148     test(AStarSearcher)
149
150 # example queries:
151 # searcher1 = Searcher(searchProblem.acyclic_delivery_problem) # DFS
152 # searcher1.search() # find first path
```

```
# searcher1.search() # find next path
# searcher2 = AStarSearcher(searchProblem.acyclic_delivery_problem) # A*
# searcher2.search() # find first path
# searcher2.search() # find next path
# searcher3 = Searcher(searchProblem.cyclic_delivery_problem) # DFS
# searcher3.search() # find first path with DFS. What do you expect to happen?
# searcher4 = AStarSearcher(searchProblem.cyclic_delivery_problem) # A*
# searcher4.search() # find first path
```

Exercise 2.2 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 2.3 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 2.4 The searcher acts like a Python iterator, in that it returns one value (here a path) and then returns other values (paths) on demand, but does not implement the iterator interface. Change the code so it implements the iterator interface. What does this enable us to do?

2.2.4 Multiple Path Pruning

To run the multiple-path pruning demo, in folder "aipython_322", load "searchMPP.py" , using e.g., `ipython -i searchMPP.py`, and copy and paste the example queries at the bottom of that file.

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

searchMPP.py — Searcher with multiple-path pruning

```
from searchGeneric import AStarSearcher, visualize
from searchProblem import Path

class SearcherMPP(AStarSearcher):
    """returns a searcher for a problem.
    Paths can be found by repeatedly calling search().
    """
    def __init__(self, problem):
        super().__init__(problem)
        self.explored = set()

    @visualize
    def search(self):
        """returns next path from an element of problem's start nodes
        to a goal node.
        Returns None if no path exists.
```

```
        """
        while not self.empty_frontier():
            path = self.frontier.pop()
            if path.end() not in self.explored:
                self.display(2, "Expanding:",path,"(cost:",path.cost,")")
                self.explored.add(path.end())
                self.num_expanded += 1
                if self.problem.is_goal(path.end()):
                    self.display(1, self.num_expanded, "paths have been expanded and",
                           len(self.frontier), "paths remain in the frontier")
                    self.solution = path # store the solution found
                    return path
                else:
                    neighs = self.problem.neighbors(path.end())
                    self.display(3,"Neighbors are", neighs)
                    for arc in neighs:
                        self.add_to_frontier(Path(path,arc))
                    self.display(3,"Frontier:",self.frontier)
        self.display(1,"No (more) solutions. Total of",
                     self.num_expanded,"paths expanded.")

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

import searchProblem
# searcherMPPcdp = SearcherMPP(searchProblem.cyclic_delivery_problem)
# print(searcherMPPcdp.search()) # find first path
```

Exercise 2.5 Implement a searcher that implements cycle pruning instead of multiple-path pruning. You need to decide whether to check for cycles when paths are added to the frontier or when they are removed. (Hint: either method can be implemented by only changing one or two lines in SearcherMPP. Hint: there is a cyle if `path.end() in path.initial_nodes()`) Compare no pruning, multiple path pruning and cycle pruning for the cyclic delivery problem. Which works better in terms of number of paths expanded, computational time or space?

2.3 Branch-and-bound Search

To run the demo, in folder "`aipython_322`", load "`searchBranchAndBound.py`", and copy and paste the example queries at the bottom of that file.

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

bound. Once a path to a goal has been found, that path is remembered as the *best_path*, the bound is reduced, and the search continues.

searchBranchAndBound.py — Branch and Bound Search

```
from searchProblem import Path
from searchGeneric import Searcher
from display import Displayable, visualize

class DF_branch_and_bound(Searcher):
    """returns a branch and bound searcher for a problem.
    An optimal path with cost less than bound can be found by calling search()
    """
    def __init__(self, problem, bound=float("inf")):
        """creates a searcher than can be used with search() to find an optimal path.
        bound gives the initial bound. By default this is infinite - meaning there
        is no initial pruning due to depth bound
        """
        super().__init__(problem)
        self.best_path = None
        self.bound = bound

    @visualize
    def search(self):
        """returns an optimal solution to a problem with cost less than bound.
        returns None if there is no solution with cost less than bound."""
        self.frontier = [Path(self.problem.start_node())]
        self.num_expanded = 0
        while self.frontier:
            path = self.frontier.pop()
            if path.cost+self.problem.heuristic(path.end()) < self.bound:
                # if path.end() not in path.initial_nodes(): # for cycle pruning
                self.display(3,"Expanding:",path,"cost:",path.cost)
                self.num_expanded += 1
                if self.problem.is_goal(path.end()):
                    self.best_path = path
                    self.bound = path.cost
                    self.display(2,"New best path:",path," cost:",path.cost)
                else:
                    neighs = self.problem.neighbors(path.end())
                    self.display(3,"Neighbors are", neighs)
                    for arc in reversed(list(neighs)):
                        self.add_to_frontier(Path(path, arc))
        self.display(1,"Number of paths expanded:",self.num_expanded,
                       "(optimal" if self.best_path else "(no", "solution found)")
        self.solution = self.best_path
        return self.best_path
```

Note that this code used *reversed* in order to expand the neighbors of a node in the left-to-right order one might expect. It does this because *pop*() removes the rightmost element of the list. The call to *list* is there because reversed only works on lists and tuples, but the neighbours can be generated.

Here is a unit test and some queries:

searchBranchAndBound.py — (continued)

```
54 from searchGeneric import test
55 if __name__ == "__main__":
56     test(DF_branch_and_bound)
57 
58 # Example queries:
59 import searchProblem
60 # searcherb1 = DF_branch_and_bound(searchProblem.acyclic_delivery_problem)
61 # print(searcherb1.search())    # find optimal path
62 # searcherb2 = DF_branch_and_bound(searchProblem.cyclic_delivery_problem, bound=100)
63 # print(searcherb2.search())    # find optimal path
```

Exercise 2.6 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 2.7 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 ^C if it goes on forever)")
48      tsearcher = Searcher(problem)
49      print("Path found:",tsearcher.search()," cost=",tsearcher.solution.cost)
50
51
52  import searchProblem
53  from searchTest import run
54  if __name__ == "__main__":
55      run(searchProblem.problem1,"Problem 1")
56  #   run(searchProblem.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 3

Reasoning with Constraints

3.1 Constraint Satisfaction Problems

3.1.1 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 list (or tuple) 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 arguments as there are variables in the scope. The condition must have a `__name__` property that gives a printable name of the function; built-in functions and functions that are defined using *def* have such a property; for other functions you may need to define this property.

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     * string: a string for printing the constraints. All of the strings must be unique.
16     for the variables
17     """
18     def __init__(self, scope, condition, string=None):
19         self.scope = scope
20         self.condition = condition
21         if string is None:
22             self.string = self.condition.__name__ + str(self.scope)
23         else:
```

```
24            self.string = string
25
26    def __repr__(self):
27        return self.string
```

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

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

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

cspProblem.py — (continued)

```
29    def holds(self,assignment):
30        """returns the value of Constraint con evaluated in assignment.
31
32        precondition: all variables are assigned in assignment
33        """
34        return self.condition(*tuple(assignment[v] for v in self.scope))
```

3.1.2 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)

```
36 class CSP(object):
37     """A CSP consists of
38     * domains, a dictionary that maps each variable to its domain
39     * constraints, a list of constraints
40     * variables, a set of variables
```

```
    * var_to_const, a variable to set of constraints dictionary
    """
    def __init__(self, domains, constraints, positions={}):
        """domains is a variable:domain dictionary
        constraints is a list of constriants
        """
        self.variables = set(domains)
        self.domains = domains
        self.constraints = constraints
        self.positions = positions
        self.var_to_const = {var:set() for var in self.variables}
        for con in constraints:
            for var in con.scope:
                self.var_to_const[var].add(con)

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

    def __repr__(self):
        """more detailed string representation of CSP"""
        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)

```
    def consistent(self,assignment):
        """assignment is a variable:value dictionary
        returns True if all of the constraints that can be evaluated
                        evaluate to True given assignment.
        """
        return all(con.holds(assignment)
                   for con in self.constraints
                   if all(v in assignment for v in con.scope))
```

3.1.3 Examples

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

cspExamples.py — Example CSPs

```
from cspProblem import CSP, Constraint
```

```
from operator import lt,ne,eq,gt

def ne_(val):
    """not equal value"""
    # nev = lambda x: x != val # alternative definition
    # nev = partial(neq,val)  # another alternative definition
    def nev(x):
        return val != x
    nev.__name__ = str(val)+"!="  # name of the function
    return nev
```

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

cspExamples.py — (continued)

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

The CSP, *csp*0 has variables X, Y and Z, each with domain $\{1, 2, 3\}$. The constraints are $X < Y$ and $Y < Z$.

cspExamples.py — (continued)

```
csp0 = CSP({'X':{1,2,3},'Y':{1,2,3}, 'Z':{1,2,3}},
          [ Constraint(['X','Y'],lt),
            Constraint(['Y','Z'],lt)])
```

The CSP, *csp*1 has variables A, B and C, each with domain $\{1, 2, 3, 4\}$. The constraints are $A < B$, $B \neq 2$ and $B < C$. This is slightly more interesting than *csp*0 as it has more solutions. This example is used in the unit tests, and so if it is changed, the unit tests need to be changed.

cspExamples.py — (continued)

```
C0 = Constraint(['A','B'], lt, "A < B")
C1 = Constraint(['B'], ne_(2), "B != 2")
C2 = Constraint(['B','C'], lt, "B < C")
csp1 = CSP({'A':{1,2,3,4},'B':{1,2,3,4}, 'C':{1,2,3,4}},
          [C0, C1, C2],
          positions={"A": (1, 0),
                     "B": (3, 0),
                     "C": (5, 0),
                     "A < B": (2, 1),
                     "B < C": (4, 1),
                     "B != 2": (3, 2)})
```

The next CSP, *csp*2 is Example 4.9 of the textbook; the domain consistent network (after applying the unary constriants) is shown in Figure 3.1.

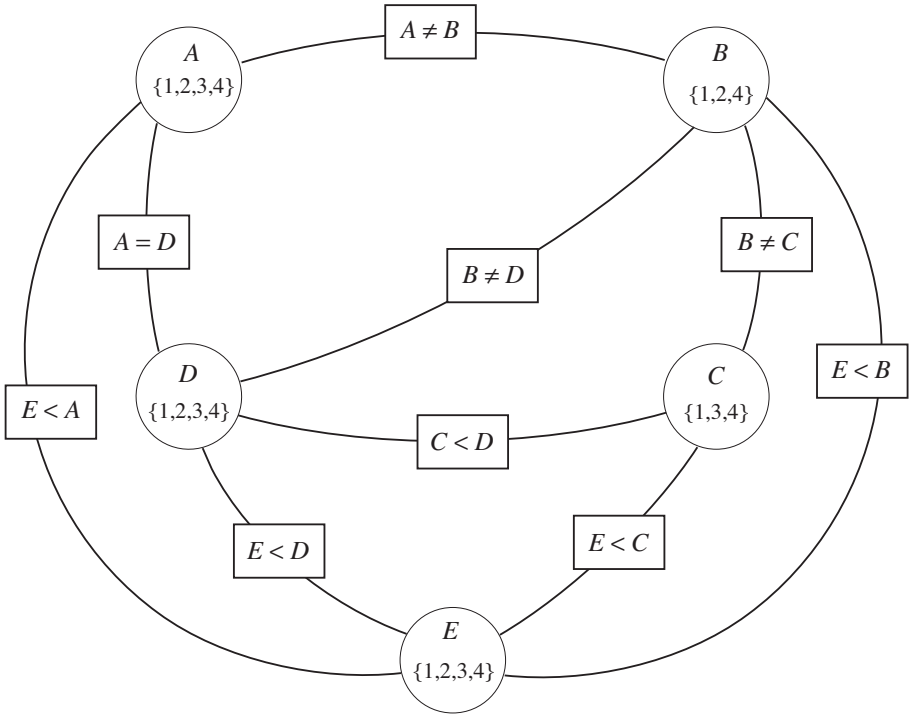


Figure 3.1: Domain-consistent constraint network (*csp*2).

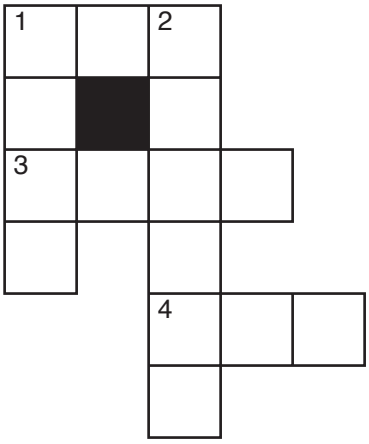
cspExamples.py — (continued)

```
48 csp2 = CSP({'A':{1,2,3,4},'B':{1,2,3,4}, 'C':{1,2,3,4},
49            'D':{1,2,3,4}, 'E':{1,2,3,4}},
50           [ Constraint(['B'], ne_(3), "B != 3"),
51            Constraint(['C'], ne_(2), "C != 2"),
52            Constraint(['A','B'], ne, "A != B"),
53            Constraint(['B','C'], ne, "A != C"),
54            Constraint(['C','D'], lt, "C < D"),
55            Constraint(['A','D'], eq, "A = D"),
56            Constraint(['A','E'], gt, "A > E"),
57            Constraint(['B','E'], gt, "B > E"),
58            Constraint(['C','E'], gt, "C > E"),
59            Constraint(['D','E'], gt, "D > E"),
60            Constraint(['B','D'], ne, "B != D")])
```

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

cspExamples.py — (continued)

```
62 csp3 = CSP({'A':{1,2,3,4},'B':{1,2,3,4}, 'C':{1,2,3,4},
63            'D':{1,2,3,4}, 'E':{1,2,3,4}},
64           [Constraint(['A','B'], ne, "A != B"),
65            Constraint(['A','D'], lt, "A < D"),
```



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

Figure 3.2: A crossword puzzle to be solved

```
66            Constraint(['A','E'], lambda a,e: (a-e)%2 == 1, "A-E is odd"), # A-E is odd
67            Constraint(['B','E'], lt, "B < E"),
68            Constraint(['D','C'], lt, "D < C"),
69            Constraint(['C','E'], ne, "C != E"),
70            Constraint(['D','E'], ne, "D != E")])
```

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

cspExamples.py — (continued)

```
72 def adjacent(x,y):
73    """True when x and y are adjacent numbers"""
74    return abs(x-y) == 1
75
76 csp4 = CSP({'A':{1,2,3,4,5},'B':{1,2,3,4,5}, 'C':{1,2,3,4,5},
77             'D':{1,2,3,4,5}, 'E':{1,2,3,4,5}},
78            [Constraint(['A','B'], adjacent, "adjacent(A,B)"),
79             Constraint(['B','C'], adjacent, "adjacent(B,C)"),
80             Constraint(['C','D'], adjacent, "adjacent(C,D)"),
81             Constraint(['D','E'], adjacent, "adjacent(D,E)"),
82             Constraint(['A','C'], ne, "A != C"),
83             Constraint(['B','D'], ne, "A != D"),
84             Constraint(['C','E'], ne, "C != E")])
```

The following examples represent the crossword shown in Figure 3.2.

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

cspExamples.py — (continued)

```
def meet_at(p1,p2):
    """returns a function of two words that is true when the words intersect at postions p1, p2.
    The positions are relative to the words; starting at position 0.
    meet_at(p1,p2)(w1,w2) is true if the same letter is at position p1 of word w1
         and at position p2 of word w2.
    """
    def meets(w1,w2):
        return w1[p1] == w2[p2]
    meets.__name__ = "meet_at("+str(p1)+','+str(p2)+')'
    return meets

crossword1 = CSP({'one_across':{'ant', 'big', 'bus', 'car', 'has'},
                  'one_down':{'book', 'buys', 'hold', 'lane', 'year'},
                  'two_down':{'ginger', 'search', 'symbol', 'syntax'},
                  'three_across':{'book', 'buys', 'hold', 'land', 'year'},
                  'four_across':{'ant', 'big', 'bus', 'car', 'has'}},
                  [Constraint(['one_across','one_down'], meet_at(0,0)),
                   Constraint(['one_across','two_down'], meet_at(2,0)),
                   Constraint(['three_across','two_down'], meet_at(2,2)),
                   Constraint(['three_across','one_down'], meet_at(0,2)),
                   Constraint(['four_across','two_down'], meet_at(0,4))])
```

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

cspExamples.py — (continued)

```
words = {'ant', 'big', 'bus', 'car', 'has','book', 'buys', 'hold',
         'lane', 'year', 'ginger', 'search', 'symbol', 'syntax'}

def is_word(*letters, words=words):
    """is true if the letters concatenated form a word in words"""
    return "".join(letters) in words

letters = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
  "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y",
  "z"]

crossword1d = CSP({'p00':letters, 'p10':letters, 'p20':letters, # first row
                   'p01':letters, 'p21':letters, # second row
                   'p02':letters, 'p12':letters, 'p22':letters, 'p32':letters, # third row
                   'p03':letters, 'p23':letters, #fourth row
                   'p24':letters, 'p34':letters, 'p44':letters, # fifth row
                   'p25':letters # sixth row
                   },
                  [Constraint(['p00', 'p10', 'p20'], is_word), #1-across
                   Constraint(['p00', 'p01', 'p02', 'p03'], is_word), # 1-down
                   Constraint(['p02', 'p12', 'p22', 'p32'], is_word), # 3-across
                   Constraint(['p20', 'p21', 'p22', 'p23', 'p24', 'p25'], is_word), # 2-down
                   Constraint(['p24', 'p34', 'p44'], is_word) # 4-across
                   ])
```

Unit tests

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

cspExamples.py — (continued)

```
133 def test(CSP_solver, csp=csp1,
134            solutions=[{'A': 1, 'B': 3, 'C': 4}, {'A': 2, 'B': 3, 'C': 4}]):
135     """CSP_solver is a solver that takes a csp and returns a solution
136     csp is a constraint satisfaction problem
137     solutions is the list of all solutions to csp
138     This tests whether the solution returned by CSP_solver is a solution.
139     """
140     print("Testing csp with",CSP_solver.__doc__)
141     sol0 = CSP_solver(csp)
142     print("Solution found:",sol0)
143     assert sol0 in solutions, "Solution not correct for "+str(csp)
144     print("Passed unit test")
```

Exercise 3.1 Modify *test* so that instead of taking in a list of solutions, it checks whether the returned solution actually is a solution.

Exercise 3.2 Propose a test that is appropriate for CSPs with no solutions. Assume that the test designer knows there are no solutions. Consider what a CSP solver should return if there are no solutions to the CSP.

Exercise 3.3 Write a unit test that checks whether all solutions (e.g., for the search algorithms that can return multiple solutions) are correct, and whether all solutions can be found.

3.2 Solving a CSP using Search

To run the demo, in folder "aipython_322", 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 which does not violate any constraints (so that dictionaries that vilate any constratints are not added).
- An arc corresponds to an assignment of a value to the next variable. This assumes a static ordering; the next variable chosen to split does not depend on the context. If no variable ordering is given, this makes no attempt to choose a good ordering.

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

```
from cspProblem import CSP, Constraint
from searchProblem import Arc, Search_problem
from utilities import dict_union

class Search_from_CSP(Search_problem):
    """A search problem directly from the CSP.

    A node is a variable:value dictionary"""
    def __init__(self, csp, variable_order=None):
        self.csp=csp
        if variable_order:
            assert set(variable_order) == set(csp.variables)
            assert len(variable_order) == len(csp.variables)
            self.variables = variable_order
        else:
            self.variables = list(csp.variables)

    def is_goal(self, node):
        """returns whether the current node is a goal for the search
        """
        return len(node)==len(self.csp.variables)

    def start_node(self):
        """returns the start node for the search
        """
        return {}
```

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

cspSearch.py — (continued)

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

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

cspSearch.py — (continued)

```
from cspExamples import csp1,csp2,test, crossword1, crossword1d
```

```
from searchGeneric import Searcher

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

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

## Test Solving CSPs with Search:
searcher1 = Searcher(Search_from_CSP(csp1))
#print(searcher1.search()) # get next solution
searcher2 = Searcher(Search_from_CSP(csp2))
#print(searcher2.search()) # get next solution
searcher3 = Searcher(Search_from_CSP(crossword1))
#print(searcher3.search()) # get next solution
searcher4 = Searcher(Search_from_CSP(crossword1d))
#print(searcher4.search()) # get next solution (warning: slow)
```

Exercise 3.4 What would happen if we constructed the new assignment by assigning $node[var] = val$ (with side effects) instead of using dictionary union? Give an example of where this could give a wrong answer. How could the algorithm be changed to work with side effects? (Hint: think about what information needs to be in a node).

Exercise 3.5 Change neighbors so that it returns an iterator of values rather than a list. (Hint: use *yield*.)

3.3 Consistency Algorithms

To run the demo, in folder "aipython_322", load "cspConsistency.py", and copy and paste the commented-out example queries at the bottom of that file.

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

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

```
from display import Displayable

class Con_solver(Displayable):
    """Solves a CSP with arc consistency and domain splitting
    """
    def __init__(self, csp, **kwargs):
        """a CSP solver that uses arc consistency
        * csp is the CSP to be solved
```

```
        * kwargs is the keyword arguments for Displayable superclass
        """
        self.csp = csp
        super().__init__(**kwargs) # Or Displayable.__init__(self,**kwargs)
```

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

cspConsistency.py — (continued)

```
    def make_arc_consistent(self, orig_domains=None, to_do=None):
        """Makes this CSP arc-consistent using generalized arc consistency
        orig_domains is the original domains
        to_do is a set of (variable,constraint) pairs
        returns the reduced domains (an arc-consistent variable:domain dictionary)
        """
        if orig_domains is None:
            orig_domains = self.csp.domains
        if to_do is None:
            to_do = {(var, const) for const in self.csp.constraints
                    for var in const.scope}
        else:
            to_do = to_do.copy() # use a copy of to_do
        domains = orig_domains.copy()
        self.display(2,"Performing AC with domains", domains)
        while to_do:
            var, const = self.select_arc(to_do)
            self.display(3, "Processing arc (", var, ",", const, ")")
            other_vars = [ov for ov in const.scope if ov != var]
            new_domain = {val for val in domains[var]
                          if self.any_holds(domains, const, {var: val}, other_vars)}
            if new_domain != domains[var]:
                self.display(4, "Arc: (", var, ",", const, ") is inconsistent")
                self.display(3, "Domain pruned", "dom(", var, ") =", new_domain,
                             " due to ", const)
                domains[var] = new_domain
                add_to_do = self.new_to_do(var, const) - to_do
                to_do |= add_to_do    # set union
                self.display(3, " adding", add_to_do if add_to_do else "nothing", "to to_do.")
            self.display(4, "Arc: (", var, ",", const, ") now consistent")
        self.display(2, "AC done. Reduced domains", domains)
        return domains

    def new_to_do(self, var, const):
        """returns new elements to be added to to_do after assigning
        variable var in constraint const.
        """
        return {(nvar, nconst) for nconst in self.csp.var_to_const[var]
                if nconst != const
                for nvar in nconst.scope
```

```
64              if nvar != var}
```

The following selects an arc. Any element of *to_do* can be selected. The selected element needs to be removed from *to_do*. The default implementation just selects which ever element *pop* method for sets returns. 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)

```
66      def select_arc(self, to_do):
67          """Selects the arc to be taken from to_do .
68          * to_do is a set of arcs, where an arc is a (variable,constraint) pair
69          the element selected must be removed from to_do.
70          """
71          return to_do.pop()
```

The value of `new_domain` is the subset of the domain of `var` that is consistemt with the assignment to the other variables. It might be easier to understand the following code, which treats unary (with no other variables in the constraint) and binary (with one other variables in the constraint) constraints as special cases (this can replace the assignment to `new_domain` in the above code):

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

any_holds is a recursive function that tries to finds an assignment of values to the other variables (*other_vars*) that satisfies constraint *const* given the assignment in *env*. The integer variable *ind* specifies which index to *other_vars* needs to be checked next. As soon as one assignment returns *True*, the algorithm returns *True*. 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)

```
73      def any_holds(self, domains, const, env, other_vars, ind=0):
74          """returns True if Constraint const holds for an assignment
75          that extends env with the variables in other_vars[ind:]
76          env is a dictionary
77          Warning: this has side effects and changes the elements of env
78          """
```

```
        if ind == len(other_vars):
            return const.holds(env)
        else:
            var = other_vars[ind]
            for val in domains[var]:
                # env = dict_union(env,{var:val}) # no side effects!
                env[var] = val
                if self.any_holds(domains, const, env, other_vars, ind + 1):
                    return True
            return False
```

3.3.1 Direct Implementation of Domain Splitting

The following is a direct implementation of domain splitting with arc consistency that uses recursion. It finds one solution if one exists or returns False if there are no solutions.

cspConsistency.py — (continued)

```
    def solve_one(self, domains=None, to_do=None):
        """return a solution to the current CSP or False if there are no solutions
        to_do is the list of arcs to check
        """
        if domains is None:
            domains = self.csp.domains
        new_domains = self.make_arc_consistent(domains, to_do)
        if any(len(new_domains[var]) == 0 for var in domains):
            return False
        elif all(len(new_domains[var]) == 1 for var in domains):
            self.display(2, "solution:", {var: select(
                new_domains[var]) for var in new_domains})
            return {var: select(new_domains[var]) for var in domains}
        else:
            var = self.select_var(x for x in self.csp.variables if len(new_domains[x]) > 1)
            if var:
                dom1, dom2 = partition_domain(new_domains[var])
                self.display(3, "...splitting", var, "into", dom1, "and", dom2)
                new_doms1 = copy_with_assign(new_domains, var, dom1)
                new_doms2 = copy_with_assign(new_domains, var, dom2)
                to_do = self.new_to_do(var, None)
                self.display(3, " adding", to_do if to_do else "nothing", "to to_do.")
                return self.solve_one(new_doms1, to_do) or self.solve_one(new_doms2, to_do)

    def select_var(self, iter_vars):
        """return the next variable to split"""
        return select(iter_vars)

def partition_domain(dom):
    """partitions domain dom into two.
    """
    split = len(dom) // 2
```

```
122     dom1 = set(list(dom)[:split])
123     dom2 = dom - dom1
124     return dom1, dom2
```

The domains are implemented as a dictionary that maps each variables to its domain. Assigning a value in Python has side effects which we want to avoid. *copy_with_assign* takes a copy of the domains dictionary, perhaps allowing for a new domain for a variable. It creates a copy of the CSP with an (optional) assignment of a new domain to a variable. Only the domains are copied.

cspConsistency.py — (continued)

```
126 def copy_with_assign(domains, var=None, new_domain={True, False}):
127     """create a copy of the domains with an assignment var=new_domain
128     if var==None then it is just a copy.
129     """
130     newdoms = domains.copy()
131     if var is not None:
132         newdoms[var] = new_domain
133     return newdoms
```

cspConsistency.py — (continued)

```
135 def select(iterable):
136     """select an element of iterable. Returns None if there is no such element.
137
138     This implementation just picks the first element.
139     For many of the uses, which element is selected does not affect correctness,
140     but may affect efficiency.
141     """
142     for e in iterable:
143         return e # returns first element found
```

Exercise 3.6 Implement of *solve_all* that is like *solve_one* but returns the set of all solutions.

Exercise 3.7 Implement *solve_enum* that enumerates the solutions. It should use Python's *yield* (and perhaps *yield from*).

Unit test:

cspConsistency.py — (continued)

```
145 from cspExamples import test
146 def ac_solver(csp):
147     "arc consistency (solve_one)"
148     return Con_solver(csp).solve_one()
149 if __name__ == "__main__":
150     test(ac_solver)
```

3.3.2 Domain Splitting as an interface to graph searching

An alternative implementation is to implement domain splitting in terms of the search abstraction of Chapter 2.

A node is domains dictionary.

cspConsistency.py — (continued)

```
152 from searchProblem import Arc, Search_problem
153
154 class Search_with_AC_from_CSP(Search_problem,Displayable):
155     """A search problem with arc consistency and domain splitting
156
157     A node is a CSP """
158     def __init__(self, csp):
159         self.cons = Con_solver(csp) #copy of the CSP
160         self.domains = self.cons.make_arc_consistent()
161
162     def is_goal(self, node):
163         """node is a goal if all domains have 1 element"""
164         return all(len(node[var])==1 for var in node)
165
166     def start_node(self):
167         return self.domains
168
169     def neighbors(self,node):
170         """returns the neighboring nodes of node.
171         """
172         neighs = []
173         var = select(x for x in node if len(node[x])>1)
174         if var:
175             dom1, dom2 = partition_domain(node[var])
176             self.display(2,"Splitting", var, "into", dom1, "and", dom2)
177             to_do = self.cons.new_to_do(var,None)
178             for dom in [dom1,dom2]:
179                 newdoms = copy_with_assign(node,var,dom)
180                 cons_doms = self.cons.make_arc_consistent(newdoms,to_do)
181                 if all(len(cons_doms[v])>0 for v in cons_doms):
182                     # all domains are non-empty
183                     neighs.append(Arc(node,cons_doms))
184                 else:
185                     self.display(2,"...",var,"in",dom,"has no solution")
186         return neighs
```

Exercise 3.8 When splitting a domain, this code splits the domain into half, approximately in half (without any effort to make a sensible choice). Does it work better to split one element from a domain?

Unit test:

cspConsistency.py — (continued)

```
188 from cspExamples import test
```

```
189 from searchGeneric import Searcher
190
191 def ac_search_solver(csp):
192     """arc consistency (search interface)"""
193     sol = Searcher(Search_with_AC_from_CSP(csp)).search()
194     if sol:
195         return {v:select(d) for (v,d) in sol.end().items()}
196
197 if __name__ == "__main__":
198     test(ac_search_solver)
```

Testing:

cspConsistency.py — (continued)

```
200 from cspExamples import csp1, csp2, csp3, csp4, crossword1, crossword1d
201
202 ## Test Solving CSPs with Arc consistency and domain splitting:
203 #Con_solver.max_display_level = 4 # display details of AC (0 turns off)
204 #Con_solver(csp1).solve_one()
205 #searcher1d = Searcher(Search_with_AC_from_CSP(csp1))
206 #print(searcher1d.search())
207 #Searcher.max_display_level = 2 # display search trace (0 turns off)
208 #searcher2c = Searcher(Search_with_AC_from_CSP(csp2))
209 #print(searcher2c.search())
210 #searcher3c = Searcher(Search_with_AC_from_CSP(crossword1))
211 #print(searcher3c.search())
212 #searcher5c = Searcher(Search_with_AC_from_CSP(crossword1d))
213 #print(searcher5c.search())
```

3.4 Solving CSPs using Stochastic Local Search

To run the demo, in folder "aipython_322", 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 structures:

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

cspSLS.py — Stochastic Local Search for Solving CSPs

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

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

    A node is a variable:value dictionary"""
    def __init__(self, csp):
        self.csp = csp
        self.variables_to_select = {var for var in self.csp.variables
                                    if len(self.csp.domains[var]) > 1}
        # Create assignment and conflicts set
        self.current_assignment = None # this will trigger a random restart
        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)

```
    def restart(self):
        """creates a new total assignment and the conflict set
        """
        self.current_assignment = {var:random_sample(dom) for
                                   (var,dom) in self.csp.domains.items()}
        self.display(2,"Initial assignment",self.current_assignment)
        self.conflicts = set()
        for con in self.csp.constraints:
            if not con.holds(self.current_assignment):
                self.conflicts.add(con)
        self.display(2,"Number of conflicts",len(self.conflicts))
        self.variable_pq = None
```

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

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

The argument *prob_anycon* is the probability that the any-conflict strategy is used (which selects a variable at random that is in a conflict), assuming that it is not picking a best variable. Note that for the probability parameters, any value less that zero acts like probability zero and any value greater than 1 acts

like probability 1. This means that when *prob_anycon* $= 1.0$, a best variable is chosen with probability *prob_best*, otherwise a variable in any conflict is chosen. A variable is chosen at random with probability $1 - prob\_anycon - prob\_best$ as long as that is positive.

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

cspSLS.py — (continued)

```
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 3.9 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*).

3.4.1 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 3.10 This makes no attempt to find the best alternative value for a variable. Modify the code so that after selecting a variable it selects a value the reduces the number of conflicts by the most. Have a parameter that specifies the probability that the best value is chosen.

3.4.2 Two-Stage Choice

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

The main complexity here is to maintain the priority queue. This uses the dictionary *var_differential* which specifies how much the values of variables should change. This is used with the updatable queue (page 55) to find a variable 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()
```

```
        pick_best_or_con = prob_best + prob_anycon
        for i in range(max_steps):
            self.number_of_steps +=1
            randnum = random.random()
            ## Pick a variable
            if randnum < prob_best: # pick best variable
                var,oldval = self.variable_pq.top()
            elif randnum < pick_best_or_con: # pick a variable in a conflict
                con = random_sample(self.conflicts)
                var = random_sample(con.scope)
            else: #pick any variable that can be selected
                var = random_sample(self.variables_to_select)
            if len(self.csp.domains[var]) > 1: # var has other values
                ## Pick a value
                val = random_sample(self.csp.domains[var] -
                                    {self.current_assignment[var]})
                self.display(2,"Assigning",var,val)
                ## Update the priority queue
                var_differential = {}
                self.current_assignment[var]=val
                for varcon in self.csp.var_to_const[var]:
                    self.display(3,"Checking",varcon)
                    if varcon.holds(self.current_assignment):
                        if varcon in self.conflicts: #was incons, now consis
                            self.display(3,"Became consistent",varcon)
                            self.conflicts.remove(varcon)
                            for v in varcon.scope: # v is in one fewer conflicts
                                var_differential[v] = var_differential.get(v,0)-1
                    else:
                        if varcon not in self.conflicts: # was consis, not now
                            self.display(3,"Became inconsistent",varcon)
                            self.conflicts.add(varcon)
                            for v in varcon.scope: # v is in one more conflicts
                                var_differential[v] = var_differential.get(v,0)+1
                self.variable_pq.update_each_priority(var_differential)
                self.display(2,"Number of conflicts",len(self.conflicts))
            if not self.conflicts: # no conflicts, so solution found
                self.display(1,"Solution found:", self.current_assignment,"in",
                             self.number_of_steps,"steps")
                return self.number_of_steps
        self.display(1,"No solution in",self.number_of_steps,"steps",
                     len(self.conflicts),"conflicts remain")
        return None
```

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

cspSLS.py — (continued)

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

Exercise 3.12 These implementations always select a value for the variable selected that is different from its current value (if that is possible). Change the code so that it does not have this restriction (so it can leave the value the same). Would you expect this code to be faster? Does it work worse (or better)?

3.4.3 Updatable Priority Queues

An **updatable priority queue** is a priority queue, where key-value pairs can be stored, and the pair with the smallest key can be found and removed quickly, and where the values can be updated. This implementation follows the idea of `http://docs.python.org/3.5/library/heapq.html`, where the updated elements are marked as removed. This means that the priority queue can be used unmodified. However, this might be expensive if changes are more common than popping (as might happen if the probability of choosing the best is close to zero).

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

cspSLS.py — (continued)

```
167 class Updatable_priority_queue(object):
```

```
    """A priority queue where the values can be updated.
    Elements with the same value are ordered randomly.

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

    def add(self,elt,val):
        """adds elt to the priority queue with priority=val.
        """
        assert val <= 0,val
        assert elt not in self.elt_map, elt
        new_triple = [val, random.random(),elt]
        heapq.heappush(self.pq, new_triple)
        self.elt_map[elt] = new_triple

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

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

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

```
218         return triple[2], triple[0] # elt, value
219
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)
```

3.4.4 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
```

3.4.5 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 3.13 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.html`). Small runtimes are inaccurate, so timeit can run the same code multiple 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:`

`//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 runtime, so you will be able to tell if there is a problem with the algorithm stopping.

Index