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

David L. Poole and Alan K. Mackworth

Version 0.8.2 of August 31, 2020.

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

The complete distribution can be downloaded from
http://artint.info/AIPython/ or from http://aipython.org

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:

```
$ ipython3 -i searchGeneric.py
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.
Testing problem 1:
7 paths have been expanded and 4 paths remain in the frontier
Path found: a --> b --> c --> d --> g
Passed unit test

In [1]: searcher2 = AStarSearcher(searchProblem.acyclic_delivery_problem) #A*

In [2]: searcher2.search()  # find first path
16 paths have been expanded and 5 paths remain in the frontier
Out[2]: o103 --> o109 --> o119 --> o123 --> r123

In [3]: searcher2.search()  # find next path
21 paths have been expanded and 6 paths remain in the frontier
Out[3]: o103 --> b3 --> b4 --> o109 --> o119 --> o123 --> r123

In [4]: searcher2.search()  # find next path
28 paths have been expanded and 5 paths remain in the frontier
Out[4]: o103 --> b3 --> b1 --> b2 --> b4 --> o109 --> o119 --> o123 --> r123

In [5]: searcher2.search()  # find next path
No (more) solutions. Total of 33 paths expanded.
```

```
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 ith function in the list is meant to add i to its argument:[1]

pythonDemo.py — Some tricky examples

```python
fun_list1 = []
for i in range(5):
    def fun1(e):
        return e+i
    fun_list1.append(fun1)

fun_list2 = []
for i in range(5):
    def fun2(e,iv=i):
        return e+iv
    fun_list2.append(fun2)

fun_list3 = [lambda e: e+i for i in range(5)]

fun_list4 = [lambda e,iv=i: e+iv for i in range(5)]

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)

```python
# in Shell do
## ipython -i pythonDemo.py
# Try these (copy text after the comment symbol and paste in the Python prompt):
# print([f(10) for f in fun_list1])
# print([f(10) for f in fun_list2])
# print([f(10) for f in fun_list3])
# 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)

```python
def myrange(start, stop, step=1):
    """enumerates the values from start in steps of size step that are
    less than stop.
    """
    assert step>0, "only positive steps implemented in myrange"
    i = start
    while i<stop:
        yield i
        i += step

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)

```python
def ga(n):
    """generates square of even nonnegative integers less than n"""
    for e in range(n):
        if e%2==0:
            yield e*e
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)

```
def myenumerate(enum):
    for i in range(len(enum)):
        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")
```

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)

```
import matplotlib.pyplot as plt

def myplot(min,max,step,fun1,fun2):
    plt.ion() # make it interactive
    plt.xlabel("The x axis")
    plt.ylabel("The y axis")
    plt.xscale('linear') # Makes a 'log' or 'linear' scale
    xvalues = range(min,max,step)
    plt.plot(xvalues,[fun1(x) for x in xvalues],
                label="The first fun")
    plt.plot(xvalues,[fun2(x) for x in xvalues], linestyle='--',color='k',
                label=fun2.__doc__) # use the doc string of the function
    plt.legend(loc="upper right") # display the legend

def slin(x):
    """y=2x+7"""
    return 2*x+7
def sqfun(x):
    """y=(x-40)^2/10-20"""
    return (x-40)**2/10-20

# Try the following:
# from pythonDemo import myplot, slin, sqfun
# import matplotlib.pyplot as plt
# myplot(0,100,1,slin,sqfun)
# plt.legend(loc="best")
# import math
# plt.plot([41+40*math.cos(th/10) for th in range(50)],
#          [100+100*math.sin(th/10) for th in range(50)])
# plt.text(40,100,"ellipse?")
# plt.xscale('log')
```

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

1.7 Utilities

1.7.1 Display

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

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

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

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

The definition of *display* is:

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

```
class Displayable(object):
    """Class that uses 'display'.
    The amount of detail is controlled by max_display_level
    """
    max_display_level = 1 # can be overridden in subclasses

    def display(self,level,*args,**nargs):
        """print the arguments if level is less than or equal to the
        current max_display_level.
        level is an integer.
        the other arguments are whatever arguments print can take.
        """
        if level <= self.max_display_level:
            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)

```
def visualize(func):
```

```
    """A decorator for algorithms that do interactive visualization.
    Ignored here.
    """
    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

```
import random

def argmax(gen):
    """gen is a generator of (element,value) pairs, where value is a real.
    argmax returns an element with maximal value.
    If there are multiple elements with the max value, one is returned at random.
    """
    maxv = float('-Infinity')    # negative infinity
    maxvals = []    # list of maximal elements
    for (e,v) in gen:
        if v>maxv:
            maxvals,maxv = [e], v
        elif v==maxv:
            maxvals.append(e)
    return random.choice(maxvals)

# Try:
# 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)

```
def flip(prob):
    """return true with probability prob"""
    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)

```
def dict_union(d1,d2):
    """returns a dictionary that contains the keys of d1 and d2.
    The value for each key that is in d2 is the value from d2,
    otherwise it is the value from d1.
    This does not have side effects.
    """
    d = dict(d1)  # copy d1
    d.update(d2)
    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)

```
def test():
    """Test part of utilities"""
    assert argmax(enumerate([1,6,55,3,55,23])) in [2,4]
    assert dict_union({1:4, 2:5, 3:4},{5:7, 2:9}) == {1:4, 2:9, 3:4, 5:7}
    print("Passed unit test in utilities")

if __name__ == "__main__":
    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

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

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

    def start_node(self):
        """returns start node"""
        raise NotImplementedError("start_node") # abstract method

    def is_goal(self,node):
        """is True if node is a goal"""
        raise NotImplementedError("is_goal") # abstract method

    def neighbors(self,node):
        """returns a list of the arcs for the neighbors of node"""
        raise NotImplementedError("neighbors") # abstract method

    def heuristic(self,n):
        """Gives the heuristic value of node n.
        Returns 0 if not overridden."""
        return 0
```

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

searchProblem.py — (continued)

```
class Arc(object):
    """An arc has a from_node and a to_node node and a (non-negative) cost"""
    def __init__(self, from_node, to_node, cost=1, action=None):
        assert cost >= 0, ("Cost cannot be negative for"+
                           str(from_node)+"->"+str(to_node)+", cost: "+str(cost))
        self.from_node = from_node
        self.to_node = to_node
        self.action = action
        self.cost=cost

    def __repr__(self):
        """string representation of an arc"""
        if self.action:
            return str(self.from_node)+" --"+str(self.action)+"--> "+str(self.to_node)
        else:
            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)

```
class Search_problem_from_explicit_graph(Search_problem):
    """A search problem consists of:
    * a list or set of nodes
    * a list or set of arcs
    * a start node
    * a list or set of goal nodes
    * a dictionary that maps each node into its heuristic value.
    * a dictionary that maps each node into its heuristic value
    """

    def __init__(self, nodes, arcs, start=None, goals=set(), hmap={}, positions={}):
        self.neighs = {}
        self.nodes = nodes
        for node in nodes:
            self.neighs[node]=[]
        self.arcs = arcs
        for arc in arcs:
            self.neighs[arc.from_node].append(arc)
        self.start = start
        self.goals = goals
        self.hmap = hmap
        self.positions = positions

    def start_node(self):
        """returns start node"""
        return self.start

    def is_goal(self,node):
        """is True if node is a goal"""
        return node in self.goals

    def neighbors(self,node):
        """returns the neighbors of node"""
        return self.neighs[node]

    def heuristic(self,node):
        """Gives the heuristic value of node n.
        Returns 0 if not overridden in the hmap."""
        if node in self.hmap:
            return self.hmap[node]
```

```
        else:
            return 0

    def __repr__(self):
        """returns a string representation of the search problem"""
        res=""
        for arc in self.arcs:
            res += str(arc)+". "
        return res
```

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

searchProblem.py — (continued)

```
    def neighbor_nodes(self,node):
        """returns an iterator over the neighbors of node"""
        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)

```
class Path(object):
    """A path is either a node or a path followed by an arc"""

    def __init__(self,initial,arc=None):
        """initial is either a node (in which case arc is None) or
        a path (in which case arc is an object of type Arc)"""
        self.initial = initial
        self.arc=arc
        if arc is None:
            self.cost=0
        else:
            self.cost = initial.cost+arc.cost

    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)

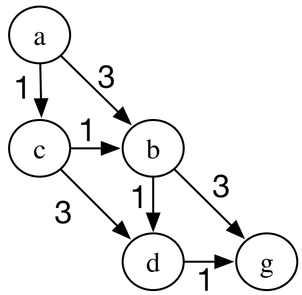

Figure 2.1: problem1

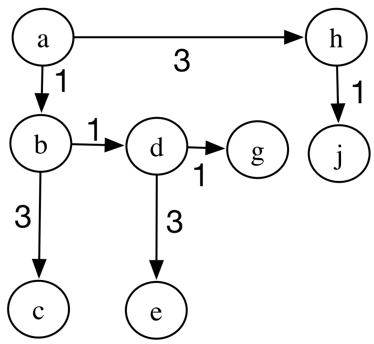

Figure 2.2: problem2

```
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)

```python
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
```

```
            if self.problem.is_goal(path.end()): # solution found
                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 reversed(list(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.")
```

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)

```
import heapq       # part of the Python standard library
from searchProblem import Path

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)

```
class AStarSearcher(Searcher):
    """returns a searcher for a problem.
    Paths can be found by repeatedly calling search().
    """

    def __init__(self, problem):
        super().__init__(problem)

    def initialize_frontier(self):
        self.frontier = FrontierPQ()

    def empty_frontier(self):
        return self.frontier.empty()

    def add_to_frontier(self,path):
        """add path to the frontier with the appropriate cost"""
        value = path.cost+self.problem.heuristic(path.end())
        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)

```
import searchProblem as searchProblem

def test(SearchClass, problem=searchProblem.problem1, solutions=[['g','d','b','c','a']] ):
    """Unit test for aipython searching algorithms.
    SearchClass is a class that takes a problemm and implements search()
    problem is a search problem
    solutions is a list of optimal solutions
    """
    print("Testing problem 1:")
    schr1 = SearchClass(problem)
    path1 = schr1.search()
    print("Path found:",path1)
    assert path1 is not None, "No path is found in problem1"
    assert list(path1.nodes()) in solutions, "Shortest path not found in problem1"
    print("Passed unit test")

if __name__ == "__main__":
    #test(Searcher)
    test(AStarSearcher)

# example queries:
# searcher1 = Searcher(searchProblem.acyclic_delivery_problem) # DFS
# 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.) 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:
                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)

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

# Example queries:
import searchProblem
# searcherb1 = DF_branch_and_bound(searchProblem.acyclic_delivery_problem)
# print(searcherb1.search())    # find optimal path
# searcherb2 = DF_branch_and_bound(searchProblem.cyclic_delivery_problem, bound=100)
# 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

```
from searchGeneric import Searcher, AStarSearcher
from searchBranchAndBound import DF_branch_and_bound
from searchMPP import SearcherMPP

DF_branch_and_bound.max_display_level = 1
Searcher.max_display_level = 1

def run(problem,name):
    print("\n\n*******",name)

    print("\nA*:")
    asearcher = AStarSearcher(problem)
    print("Path found:",asearcher.search()," cost=",asearcher.solution.cost)
    print("there are",asearcher.frontier.count(asearcher.solution.cost),
          "elements remaining on the queue with f-value=",asearcher.solution.cost)

    print("\nA* with MPP:"),
    msearcher = SearcherMPP(problem)
    print("Path found:",msearcher.search()," cost=",msearcher.solution.cost)
    print("there are",msearcher.frontier.count(msearcher.solution.cost),
          "elements remaining on the queue with f-value=",msearcher.solution.cost)

    bound = asearcher.solution.cost+0.01
```

```
    print("\nBranch and bound (with too-good initial bound of", bound,")")
    tbb = DF_branch_and_bound(problem,bound) # cheating!!!!
    print("Path found:",tbb.search()," cost=",tbb.solution.cost)
    print("Rerunning B&B")
    print("Path found:",tbb.search())

    bbound = asearcher.solution.cost*2+10
    print("\nBranch and bound (with not-very-good initial bound of", bbound, ")")
    tbb2 = DF_branch_and_bound(problem,bbound) # cheating!!!!
    print("Path found:",tbb2.search()," cost=",tbb2.solution.cost)
    print("Rerunning B&B")
    print("Path found:",tbb2.search())

    print("\nDepth-first search: (Use ^C if it goes on forever)")
    tsearcher = Searcher(problem)
    print("Path found:",tsearcher.search()," cost=",tsearcher.solution.cost)


import searchProblem
from searchTest import run
if __name__ == "__main__":
    run(searchProblem.problem1,"Problem 1")
#   run(searchProblem.acyclic_delivery_problem,"Acyclic Delivery")
#   run(searchProblem.cyclic_delivery_problem,"Cyclic Delivery")
# also test some graphs with cycles, and some with multiple least-cost paths
```

Index