Search Exercise 2

Defining a Search Problem and using a General Search Algorithm

The Jug Pouring Puzzle Example

Last edit: 10am, Tuesday 31st Jan 2023

This Python notebook will introduce you to a simple software tool that will enable you to define search problems and attempt to find solutions using a variety of different algorithms parameters.

Learning Objectives

Studying and experimenting with this notebook should help you to acquire the following knowledge and skills:

- Understanding of the general structure and components of a search problem.
- Understanding of how a real problem can be specified in this form.
- Appreciation of how features of a particular search problem can affect the performance of search algorithms and how this varies depending on the algorithm.
- Ability to experiment with applying a variety of algorithms to a problem.

Using this Jupyter/Colab Notebook

This module does not involve fully-fledged programming assignments. However, we will be exploring AI topics using some software tools and code examples, that you will need to read through and modify in various ways in order to develop your understanding.

Notebook programs are a convenient way to combine formated explanatory material and code within a single file. For Python, the Jupyter notebook interface is very popular. It is based on the *Iron Python NoteBook* files format (file extension `.ipynb`). These files can also be run on the cloud using services such as Google Colab.

Quickstart use of the notebook

If you are looking at this for the first time then you will be seeing it in the Colab programming interface. You can simply proceed by reading each section and following the instructions to run the *code cells* that are interleaved with the textual information.

Futher use of this and other notebook files

After you have gone through this file and experimented with what it can do you should find out a bit more about Python Notebook files and how to use them. I have put information about his on a web page: PythonNotebookInfo. That page has links to a lot of further information which some of you may find useful. For this course you only need a basic understanding of Python and Notebook files, so looking at the extra stuff is optional; but it does include a couple of introductory videos that you might find useful.

Brandon's Generic Search Algorithm

The best way to learn about how to employ AI search algorithms on actual problems is to try it yourself, starting with some simple cases. To do this you can use my Python implementation of a general search algorith implemented in a very similar way to what Russell and Norvig describe.

When you run the following code cell it should download the my Python module from the web and install it locally. This should work on pretty much any Python platform. Obviously you will need internet access, but once it has been installed you could run the rest of the notebook off line.

Running a code cell

The following chunk of code is a *code cell*. It is actually a sequence of Linux commands that get down my code file from the web and save it on the file system of whatever system is running this file (can either be a cloud system like Colab, or your local machiine). Most code cells are Python code but in this case we are using the `!` at the beginning of the line, which enables operating system shell commands to be executed.

To run the code cell in Colab you click the run symbol (white triangle on a black circle) that will appear in the top left of the cell when you move your mouse over it. **Do that now.**

```
!echo Installing bbSearch module from web ...
!echo creating bbmodcache subfolder
!mkdir -p bbmodcache
!echo downloading bbSearch module
!curl http://bb-ai.net.s3.amazonaws.com/bb-python-modules/bbSearch.py > bbmodcache/bbSearc
```

```
Installing bbSearch module from web ...
creating bbmodcache subfolder
downloading bbSearch module
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 18767  100 18767    0     0  51842      0 --:--:-- --:--:-- --:--:-- 51842
```

Now you should be able to `import` from the `bbSearch` module

Now try running the follwing cell. Click the run symbol (or you can use `<shift>+<enter>` as a keyboard shortcut):

```
from bbmodcache.bbSearch import SearchProblem, search
```

The bbSearch Python module should now be loded which will enable us to use the `SearchProblem` class and the `search` function.

The SearchProblem Class

Below is the code that defines the `SearchProblem` class in `bbSearch`. This is for viewing not editing. It is just a copy of what is in the actual `bbSearch` module. This class definition is essentially a template for defining search problems. It is a *base class* that needs to be *extended* to define an actual search problem, as we shall see shortly below. Hence most of the method definitions are just place holders with comments describing what they should do. But please **read the following code carefully**, paying particular attention to the functions: `__init__`, `possible_actions`, `successor` and `goal_state`.

```
class SearchProblem:

    def __init__( self ):
        """
        The __init__ method must set the initial state for the search.
        Arguments could be added to __init__ and used to configure the
        initial state and/or other aspects of a problem instance.
        """
        self.initial_state = None ## Change this to set the actual initial state!
        raise NotImplementedError

    def info(self):
        """
        This function is called when the search is stared and should
        print out useful information about the problem.
        """
        print("This is the general SearchProblem parent class")
        print("You must extend this class to encode a particular search problem.")

    def possible_actions(self, state):
        """
        This takes a state as argument and must return a list of actions.
        Both states and actions can be any kinds of python data type (e.g.
        numbers, strings, tuples or complex objects of any class).
        """
```

```
        return []

    def successor(self, state, action):
        """
        This takes a state and an action and returns the new state resulting
        from doing that action in that state. You can assume that the given
        action is in the list of 'possible_actions' for that state.
        """
        return state

    def goal_test(self, state):
        """
        This method should return True or False given any state. It should return
        True for all and only those states that are considert "goal" states.
        """
        return False

    def cost(self, path, state):
        """
        This is an optional method that you only need to define if you are using
        a cost based algorithm such as "uniform cost" or "A*". It should return
        the cost of reaching a given state via a given path.
        If this is not defined, it will is assumed that each action costs one unit
        of effort to perform, so it returns the length of the path.
        """
        return len(path)

    def heuristic(self, state):
        """
        This is an optional method that should return a heuristic value for any
        state. The value should be an estimate of the remaining cost that will be
        required to reach a goal. For an "admissible" heuristic, the value should
        always be equal to or less than the actual cost.
        """
        raise NotImplementedError

    def display_action(self, action):
        """
        You can set the way an action will be displayed in outputs.
        """
        print(action)

    def display_state(self, state):
        """
        You can set the way a state will be displayed in outputs.
        """
```

```
        print(state)

    def display_state_path( self, actions ):
        """
        This defines output of a solution path when a list of actions
        is applied to the initial state. It assumes it is a valid path
        with all actions being possible in the preceeding state.
        You probably don't need to override this.
        """
        s = self.initial_state
        self.display_state(s)
        for a in actions:
            self.display_action(a)
            s = self.successor(s,a)
            self.display_state(s)
```

A Simple Search Example: Jug Pouring Puzzle

Many simple examples are provided by types of problem that we consider to be *puzzles*. Although puzzles are primarily considers as amusements, the problems involved in solving them are of the same general form those that arise in serious practical problems. Solving such problems can certainly certainly make life easer and in some cases can be very profitable or even save lives. But puzzles are often simpler and clearer in terms of the exact details of what is the problem and what solition is required.

A typical and quite well know type of puzzle is the **Jug Pouring Puzzle** (or water pouring puzzle). There are many versions of this, but the basic idea is that we have some containers that can measure certain definite quantities of liquid and we want to measure some other quantity by transfering the liquid between the containers by a sequence of pouring actions. In order to ensure accuracy of the measurement, we are only allowed to carry out the following kinds of pouring action:

- Pour liquid from a source jug to a receiving jug until either:
    - the receiving jug becomes full (whatever cannot fit remains in the source jug);
    - or, all liquid from the source jug is transferred into the receiving jug (so the source becomes empty).

You can consult Wikipedia for further explanation: Wikipedia: Water Pouring Puzzle. An example of this kind of problem is the "Die Hard 3" Problem, which featured in the *Die Hard 3* movie. (In the movie, they have two empty jugs that they can fill from a fountain. But you can model the fountain as a container that is much bigger than the jugs.) For a theoretical consideration you could take a look at the paper *Measuring with Jugs* (Boldi, Santini, Vigna, 2002), but you will see that the hard-core mathematical treadment of this kind of problem is extremely complex.

Jug Pouring Puzzle state representation

To model this kind of puzzle as a search puzzle, we need to decide upon a suitable representation of a state. There are countless possibilities of how we can do this. Some will be more convenient or computationally efficient than others. A simple idea that works well in Python is to use a dictionry where we have a key for each of the jugs whose value is a tuple giving the maximum capacity of the jug and its current contents. So this is an example state:

```
{"small":(3,1), "medium":(5,0), "large":  (8,8) }
```

Here the "small" jug has capacity 3 units and contains 1 unit of liquid, the "medium" jug has capacity 5 units and is empty and the "large" jug has capacity 8 and is full.

`JugPouringPuzzle` Class definition

The following class definition specified `JugPouringPuzzle` as and extension of `SearchProblem`. It fills in the specific details of the methods of the template class `SearchProblem`.

You should read the code carefully to get a good understanding of how to model a search problem. The code in each of the methods is not especially complex. It is the combination of the methods and how they are used in the search algorithm that creates a very powerful algorithm.

One thing that you should note is the use of the `deepcopy` function within the `successor` method, which the starting point for creating the next state. The effect of `deepcopy(x)` is to return an object that is identical to the object referred to by `x`, except that it is a different object. This means that we can then modify the copy without changing the original object from which it was derived. This is essential in the `successor` function because if there are several actions possible from a given state we may need to generate several successors of that state. So we don't want the successor function to alter it. Forgetting this is a common error in writing search algorithms and can be very hard to debug if you are not aware of this kind of issue.

Apart from that, the rest of the code is just quite simple tests and operations on the dictionary datastructure we have specified to represent the state of the jugs.

The following chunk of code defining `JugPouringPuzzle` is in an editable and runable Python code cell, so you could edit it to modify the class definition. But for the time being you should not alter it.

```
from copy import deepcopy

class JugPouringPuzzle( SearchProblem ):

    def __init__( self, initial_state, goal_quantity ):
        """
        Initialise a JugPouringPuzzle instance by setting the initial state
        and the goal quantity.
        """
```

```
        self.initial_state = initial_state
        self.goal_quantity = goal_quantity
        print( "Creating SearchProblem object" )
        print( "Setting initial state to:", self.initial_state )
        print( "Goal quantity to measure:", self.goal_quantity )

        self.jugs = self.initial_state.keys()

    def info(self):
        print( "Measuring Jugs problem:" )
        print( "You have a number of jugs with a certain volume and initial contents, as f
        for jar in self.initial_state:
          print(f"{jar:>10} : volume={self.initial_state[jar][0]}, "
                          f"contents={self.initial_state[jar][1]}")
        print( "The goal quantity is:", self.goal_quantity)

    def possible_actions(self, state):
        """
        Possible actions are to pour liquid from one jug to another. These are represented
        (j1, j2), where j1 is the source jug and j2 is the receiver.
        j1 cannot be empty and j2 cannot be full.
        """
        actions = [] # start with empty action list
        for j1 in self.jugs:
            for j2 in self.jugs:
                if ( j1!=j2  #different jugs
                     and state[j1][1] > 0  # j1 not empty
                     and state[j2][0] > state[j2][1] # j2 not full
                   ): actions.append((j1,j2)) # action possible, add to actions list
        return actions

    def successor(self, state, action):
        """
        Give the state resulting from given state when an action (j1,j2) is formed.
        """

        ## We need to construct a new state.
        ## First make a copy of the original state.
        ## Note don't directly change the given state as the algorithm may need to
        ## construct other copies of it resulting from different possible actions
        ## Assign new_state to a deep copy of state:
        new_state = deepcopy(state)

        # The acion specifies two jugs j1 and j2
        j1, j2 = action[0], action[1]

        # Get contents of the jugs and space left in receiving jug
        vol_in_j1 = state[j1][1]
        vol_in_j2 = state[j2][1]
        space_in_j2 = state[j2][0] - state[j2][1]

        if vol_in_j1 <= space_in_j2:     # Case where all j1 contents can go into j2
            new_j1 = 0
            new_j2 = vol_in_j1 + vol_in_j2
        else:                                  # Case where only some can be transfered
```

```
            new_j1 = vol_in_j1 - space_in_j2  # Fill the space taking from j1
            new_j2 = state[j2][0]             # j2 will now be full

        ## Now add the new values for each jug to the new state dictionary
        ## (The capacity (first number) stays the same, just contents of j1 and j2 changes
        new_state[j1] = (state[j1][0], new_j1)
        new_state[j2] = (state[j2][0], new_j2)
        return new_state

    def goal_test(self, state):
        """
        Test whether the state is a goal state.
        This is when the content of one of the jugs is the same as the goal quantity.
        """
        quantities = [ state[k][1] for k in state]
        return  self.goal_quantity in quantities

    def display_action( self, action ):
        """
        Display an action in an easy to understand way.
        """
        j1, j2 = action
        print(f"Pour from {j1} to {j2}:")

    def display_state( self, state ):
        "Display a state in a nice way."
        jug_quantities = [ f"{k} ({state[k][0]}): {state[k][1]}" for k in state]
        print( ", ".join(jug_quantities))
```

Run `JugPoruingPuzzle` class definition code cell

Once you have read through the above code cell you should run it to execute the class definition (press the triangle in circle symbol at top left of the cell).

Create a `JugPouringPuzzle` instance

It is now very easy to define a specific jug pouring puzzle as an instance of the `JugPouringPuzzle` classe. For example, the following code creates a puzzle instance where we have empty small and medium jugs withe capacities 3 and 8 and a full large jug with capacity 8. The goal is to measure 4 units of liquid.
By now you should recognise that the following is a code cell and know how to run it.
In the rest of this notebook and in following exercises I will not remind you about running the code cells. You should always do that.

Note: If you forgot to run one of the previous cells you will get an error. Usually the code cells of a notebook need to be run from top to bottom in order for them to work. If you get an error such as `'xyz' is not defined` this means you probably missed one and need to go back and run previous cells. In Colab you will see a green tick mark on the left of a cell after it has been run.

```
JPP_1 = JugPouringPuzzle(
           {"small":(3,0), "medium":(5,0), "large":  (8,8) }, #initial state
           4   # goal measurement
         )
```

```
Creating SearchProblem object
Setting initial state to: {'small': (3, 0), 'medium': (5, 0), 'large': (8, 8)}
Goal quantity to measure: 4
```

The `search` function

We can now call the search function itself. This takes 3 required paremeters and some optional arguments:

- A `SearchProblem` object (for example an instance of the `JugsSearchProblem` that we have just defined.
- A chosen **search type** which can be one of:
  - `'BF/FIFO'` Breadth-First search (uses a First in First out **queue** to store generated states waiting to be tested).
  - `'DF/LIFO'` Depth-First (uses a Last In First Out **stack** to store generated states waiting to be tested).
- A **node limit** (an `int`), which is the maximum number of nodes that will be created in the search tree before the search is aborted. (Note that each node corresponds to a state, but states can potentially occur at many nodes as there could be many ways to get to the same state.)
- Optional arguments:
  - `loop_check` You can have this on or off by adding `loop_check=True` or `loop_check=False`. The default value is `False`.
  - `randomise` (`=True` or `=False`, default `False`). If this is set to true, the list of possible actions will be shuffled into a random order before they are inserted into the search queue. (If cost and heuristic functions are set they will still be used to determine position of insertion in the queue so will only make a difference for states whose cost and/or heuristic values are tied.)
  - `show_path` (`=True` or `=False`, default `True`), choose whether to output the action path found by the search.
  - `show_state_path` (`=True` or `=False`, default `False`), choose whether to output the sequence of states found by the search.
  - `return_info` (`=True` or `=False`, default `False`), If `True` the `search` function will return a dictionary giving full details of the search that was carried out and the result obtained. If `False` the function will just return a string describing whether the search was successful.

```
search( JPP_1, 'BF/FIFO', 2000, loop_check=False, randomise=False, show_state_path=True)
```

```
Measuring Jugs problem:
You have a number of jugs with a certain volume and initial contents, as follows:
     small : volume=3, contents=0
    medium : volume=5, contents=0
     large : volume=8, contents=8
The goal quantity is: 4

** Running Brandon's Search Algorithm **
Strategy: mode=BF/FIFO, cost=None, heuristic=None
Max search nodes: 2000  (max number added to queue)
Searching (will output '.' each 1000 goal_tests)

:-)) *SUCCESS* ((-:

Path length = 6
Goal state is:
small (3): 3, medium (5): 4, large (8): 1
The action path to the solution is:
Pour from large to medium:
Pour from medium to small:
Pour from small to large:
Pour from medium to small:
Pour from large to medium:
Pour from medium to small:

The state/action path to the solution is:
small (3): 0, medium (5): 0, large (8): 8
Pour from large to medium:
small (3): 0, medium (5): 5, large (8): 3
Pour from medium to small:
small (3): 3, medium (5): 2, large (8): 3
Pour from small to large:
small (3): 0, medium (5): 2, large (8): 6
Pour from medium to small:
small (3): 2, medium (5): 0, large (8): 6
Pour from large to medium:
small (3): 2, medium (5): 5, large (8): 1
Pour from medium to small:
small (3): 3, medium (5): 4, large (8): 1

SEARCH SPACE STATS:
Total nodes generated          =     1267  (includes start)
```

Changing the Initial and Goal states

The `JugPouringPuzzle` class can

Here is a very simple case. I want to measure 3 units of water. I have an empty mug an empty kettle and a sink half full of water. So the initial state is:

```
{'mug':(1,0), 'kettle':(6,0), 'sink':(40,20)},
```

Or perhaps I have a large barrel of beer and want to transfer 20 pints into a flagon, but I only