Search Exercise 6

The 8-Puzzle

The 8-Puzzle is a simplified version of the better known 15-Puzzle invented by Noyes Palmer Chapman around 1874. Please refer to the Wikipedia page or other publically available sources for general information about the puzzle.

In the 8-Puzzle we have 8 sliding tiles that can slide within in a 3x3 grid, such that there is always one empty grid position, into which any of the neighbouring tiles can slide. The aim of the puzzle is to go from an intial randomised configuration of tiles to a specified end configuration, which typically has the tiles ordered in increasing order left to right and top to bottom, with the space being in at the bottom right.

There is an interactive version with cheezy pictures that you can try here and an interactive 8-Puzzle search algorithm demo can be found here. (This has a nice visualisation of the tree structure of the search spaces for several different algorithms.)

Investigating search algorithm performance on the 8-Puzzle

As with the other exercises, we start by installing `bbSearch` and importing the `SearchProblem` class and the `search` function:

```
!mkdir -p bbmodcache
!curl http://bb-ai.net.s3.amazonaws.com/bb-python-modules/bbSearch.py > bbmodcache/bbSearch.py
from bbmodcache.bbSearch import SearchProblem, search
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 18767  100 18767    0     0  54397      0 --:--:-- --:--:-- --:--:-- 54239
Loading bbSearch Version 2.1 (at 12:45, Tue 01 Mar)
Last module source code edit 9am Thursday 24th Feb 2022
```

Representing a tile layout

It is easy to represent the layout of tiles in a state by a list of lists structure. More precisely, we have a list of three lists representing each row; and each row list contains three numbers. We will use 0 to represent the empty space (into which neighbouring tiles may slide). Hence, we can define some layouts as follows:

[ ] ↳ *已隐藏 1 个单元格*

A function for finding the position of a given number in a layout

The following simple function will be useful. I will use it to find the position of the space (0), and **it will also be useful in defining heuristics**.

```
def number_position_in_layout( n, layout):
        for i, row in enumerate(layout):
                for j, val in enumerate(row):
                        if val==n:
                                return (i,j)
```

Representing a state of the 8-Puzzle problem

One could represent any state of the 8-puzzle simply by its tile layout. This contains all the information one may need to determine possible moves and to test whether it is a goal state.

However, there is a drawback with using that representation. To calculate possible moves we need to know the position of the space, and although this is not difficult, it will take some computational effort to loop through the rows and columns of the layout representation to find the space. And if we do that thousands or even millions of times (as we might while searching a big search space), it could be computationally expensive. Hence, I represent an eight puzzle staid by a tuple of the form, `((row,col), layout)` where `(row,col)` gives the position of the space and `layout` is a list of lists representing the full layout, as specified above.

Define the `EightPuzzle` class

As in previous examples we can define (by extending Brandon's `SearchProblem` class template) a class that specifies the structure of the 8-Puzzle in a way that can be handled by a variety of different search algorithms.

The following brief notes will help you to understand the class definition in the next code cell:

- Coordinates on the tile grid are given in the form `(row,column)`
- The `action_dict` lists all possibles moves for each possible coordinate position of where the space occurs in the grid. So, for example, if the space is at `(0,0)` we can either move the tile from `(1,0)` up, or move the tile from `(1,0)` left. Specifying it like this should (I expect) be faster than computing possible moves using a function (which might be used many many times during a big search).
- In `action_dict`, actions are specified by a tuple of the form `(row, col, d)` giving the row and column of the moving piece and the direction in which it moves. (The direction `d` could be calculated from `(row,col)` for any given board state, since we would know the possition of the space into which the moving tile moves, but it will make actions and movement paths easier to interpret if we add the direction to the action representation).

- The full representation for an action, used by the algorithm takes the form `(row, col, (n, d))`. Here `n` is the number of the tile being moved and gets added to the action representation by the `possible_actions` function when it generates the possible actions from a given state. (Again this is not strictly necessary for the algorithm to work but make it much easier to interpret the sequences of moves it generates.)

```
from copy import deepcopy

class EightPuzzle( SearchProblem ):

    action_dict = {
            (0,0) : [(1, 0, 'up'),          (0, 1, 'left')],
            (0,1) : [(0, 0, 'right'), (1, 1, 'up'),          (0, 2, 'left')],
            (0,2) : [(0, 1, 'right'), (1, 2, 'up')],
            (1,0) : [(0, 0, 'down'),     (1, 1, 'left'),     (2, 0, 'up')],
            (1,1) : [(1, 0, 'right'), (0, 1, 'down'),     (1, 2, 'left'), (2, 1, 'up
            (1,2) : [(0, 2, 'down'),     (1, 1, 'right'), (2, 2, 'up')],
            (2,0) : [(1, 0, 'down'),     (2, 1, 'left')],
            (2,1) : [(2, 0, 'right'), (1, 1, 'down'),     (2, 2, 'left')],
            (2,2) : [(2, 1, 'right'), (1, 2, 'down')]
    }


    def __init__(self, initial_layout, goal_layout ):
            pos0 = number_position_in_layout( 0, initial_layout )
            # Initial state is pair giving initial position of space
            # and the initial tile layout.
            self.initial_state = ( pos0, initial_layout)
            self.goal_layout      = goal_layout


    ### I just use the position on the board (state[0]) to look up the
    ### appropriate sequence of possible actions.
    def possible_actions(self, state ):
            actions =    EightPuzzle.action_dict[state[0]]
            actions_with_tile_num = []
            for r, c, d in actions:
                    tile_num = state[1][r][c] ## find number of moving tile
                    # construct the action representation including the tile number
                    actions_with_tile_num.append( (r, c, (tile_num,d)) )
            return actions_with_tile_num

    def successor(self, state, action):
            old0row, old0col    =    state[0]        # get start position
            new0row, new0col, move = action    # unpack the action representation
            moving_number, _ = move
            ## Make a copy of the old layout
            newlayout = deepcopy(state[1])
            # Swap the positions of the new number and the space (rep by 0)
            newlayout[old0row][old0col] = moving_number
            newlayout[new0row][new0col] = 0
            return ((new0row, new0col), newlayout )
```

```
    def goal_test(self,state):
        return state[1] == self.goal_layout

    def display_action(self, action):
        _,_, move = action
        tile_num, direction = move
        print("Move tile", tile_num, direction)

    def display_state(self,state):
        for row in state[1]:
            nums = [ (n if n>0 else '.') for n in row]
            print( "  ", nums[0], nums[1], nums[2] )
```

Constructing a problem instance

We can now create a `EightPuzzle` instance by giving its initial layout and goal layout:

```
EP = EightPuzzle( LAYOUT_1, NORMAL_GOAL )
```

Running the `search` function

Now we can try running the search function. Let's start with a breadth first search. It should be able to fund an optimal path to the solution:

```
search(EP, 'BF/FIFO', 1000000, loop_check=False, show_state_path=True)
```

```
    This is the general SearchProblem parent class
    You must extend this class to encode a particular search problem.

    ** Running Brandon's Search Algorithm **
    Strategy: mode=BF/FIFO, cost=None, heuristic=None
    Max search nodes: 1000000  (max number added to queue)
    Searching (will output '.' each 1000 goal_tests)
    ................................................................................
    ................................................................................
    ................................................................................
    ..........................................................
    !! Search node limit (1000000) reached !!
    ): No solition found :(


    SEARCH SPACE STATS:
    Total nodes generated          =  1000001  (includes start)
    Nodes tested (by goal_test)    =   358301  (all expanded)
    Nodes left in queue            =   641699

    Time taken = 72.5848 seconds

    'NODE_LIMIT_EXCEEDED'
```

Hmm... What happened?

Maybe we should try something else. How about the `loop_check` option?

Using Heuristics

Another way to help the search algorithm work more effectively is of course by using a heuristic to guide the selection of moves to explore while searching for a solution.

In the notes (and in Russell and Norvig) you will have seen mention of the *misplaced tiles* heuristic and the *manhattan distance* heuristic. You should know enough to be able to implement these your self. You just need to define a function that takes a state as its argument and returns the corresponding value of the heuristic applied to that state. (Remember that the state representation consists of the space position and the tile layout, as described above.)

But to save you the trouble for the time being and let you see the effect of these heuristics, I have pre-defined them for you in my Python module `crazy8heuristics`. The following cell should install that module and import functions for the heuristics `bb_misplaced_tiles` and `bb_manhattan`:

```
!mkdir  -p  bbmodcache
!curl  http://bb-ai.net.s3.amazonaws.com/bb-python-modules/crazy8heuristics.py  >  bbmodcache/crazy
from  bbmodcache.crazy8heuristics  import  bb_misplaced_tiles,  bb_manhattan
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  4731  100  4731    0     0  16776      0 --:--:-- --:--:-- --:--:-- 16776
```

Note that the heuristics defined in `craz8heuristics` are based on the assumption that the goal is to get to a state with the `NORMAL_GOAL` tile layout, as specified above.

Running `search` with a heuristics

We can now add one of the heuristics as an option to the `search` function, as follows:

```
search(EP,  'BF/FIFO',  1000000,  heuristic=bb_misplaced_tiles,
            loop_check=True,  show_state_path=True)
```

```
This is the general SearchProblem parent class
You must extend this class to encode a particular search problem.

** Running Brandon's Search Algorithm **
Strategy: mode=BF/FIFO, cost=None, heuristic=bb_misplaced_tiles
Max search nodes: 1000000  (max number added to queue)
Searching (will output '.' each 1000 goal_tests)

:-)) *SUCCESS* ((-:

Path length = 38
Goal state is:
    1 2 3
    4 5 6
    7 8 .
The action path to the solution is:
Move tile 3 right
Move tile 6 right
Move tile 2 down
Move tile 4 left
Move tile 8 left
Move tile 3 up
Move tile 6 right
Move tile 8 down
Move tile 1 down
Move tile 7 left
Move tile 3 up
Move tile 6 up
Move tile 8 right
Move tile 2 right
Move tile 4 down
Move tile 5 down
Move tile 7 left
Move tile 1 up
Move tile 5 right
Move tile 7 down
Move tile 1 left
Move tile 5 up
Move tile 7 right
Move tile 4 up
Move tile 2 left
Move tile 7 down
Move tile 5 down
Move tile 1 right
Move tile 4 up
Move tile 2 up
Move tile 7 left
Move tile 5 down
Move tile 2 right
Move tile 4 down
Move tile 1 left
Move tile 2 up
Move tile 5 up
Move tile 8 left

The state/action path to the solution is:
    5 1 7
```

```
    5 1 7
    2 4 8
    6 3 .
Move tile 3 right
    5 1 7
    2 4 8
    6 . 3
Move tile 6 right
    5 1 7
    2 4 8
    . 6 3
Move tile 2 down
    5 1 7
    . 4 8
    2 6 3
Move tile 4 left
    5 1 7
    4 . 8
    2 6 3
Move tile 8 left
    5 1 7
    4 8 .
    2 6 3
Move tile 3 up
    5 1 7
    4 8 3
    2 6 .
Move tile 6 right
    5 1 7
    4 8 3
    2 . 6
Move tile 8 down
    5 1 7
    4 . 3
    2 8 6
Move tile 1 down
    5 . 7
    4 1 3
    2 8 6
Move tile 7 left
    5 7 .
    4 1 3
    2 8 6
Move tile 3 up
    5 7 3
    4 1 .
    2 8 6
Move tile 6 up
    5 7 3
    4 1 6
    2 8 .
Move tile 8 right
```

You should also try using the `bb_manhattan` heursitic and compare the results.

Heuristic Evaluation Exercise

Can you answer the following questions?

- Were solution path sequences found?
- If so were they good solutions?
- Did the heuristics improve performance?
- Was one better than the other?

What about using a cost function ?

Another way we can guide the way a search space is explored is by using a cost function that gives some measure of the cost of getting to a given state by going along a given path from the start point of the search.

But in the 8-Puzzle all moves are just sliding one tile into a space, so we would probably consider all moves to have equal cost. Hence, the cost of getting to any state along any path would just be the length of the path.

Well, at least this is easy to implement, as we see below. But is it useful? Let us try it and let us compare what happens with the cost function set with what happens without any cost function:

```
def cost(path, state):
    return len(path)
```

```
search(EP, 'BF/FIFO', 100000, cost=cost, heuristic=bb_manhattan, loop_check=True, show_state_
#search(EP, 'BF/FIFO', 100000, loop_check=True, show_state_path=True)
```

```
This is the general SearchProblem parent class
You must extend this class to encode a particular search problem.

** Running Brandon's Search Algorithm **
Strategy: mode=BF/FIFO, cost=cost, heuristic=bb_manhattan
Max search nodes: 100000  (max number added to queue)
Searching (will output '.' each 1000 goal_tests)
.......
:-)) *SUCCESS* ((-:

Path length = 28
Goal state is:
    1 2 3
    4 5 6
    7 8 .
Cost of reaching goal: 28
The action path to the solution is:
Move tile 3 right
Move tile 4 down
Move tile 2 right
Move tile 6 up
Move tile 4 left
Move tile 3 left
Move tile 8 down
Move tile 2 right
Move tile 6 right
Move tile 5 down
Move tile 1 left
Move tile 7 left
Move tile 2 up
Move tile 6 right
Move tile 3 up
Move tile 8 left
Move tile 6 down
Move tile 3 right
Move tile 7 down
Move tile 2 left
Move tile 3 up
Move tile 6 up
Move tile 8 right
Move tile 7 down
Move tile 5 right
Move tile 4 up
Move tile 7 left
Move tile 8 left

The state/action path to the solution is:
    5 1 7
    2 4 8
    6 3 .
Move tile 3 right
    5 1 7
    2 4 8
    6 . 3
Move tile 4 down
    5 1 7
    2 . 8
```

```
    2 . 8
    6 4 3
Move tile 2 right
    5 1 7
    . 2 8
    6 4 3
Move tile 6 up
    5 1 7
    6 2 8
    . 4 3
Move tile 4 left
    5 1 7
    6 2 8
    4 . 3
Move tile 3 left
    5 1 7
    6 2 8
    4 3 .
Move tile 8 down
    5 1 7
    6 2 .
    4 3 8
Move tile 2 right
    5 1 7
    6 . 2
    4 3 8
Move tile 6 right
    5 1 7
    . 6 2
    4 3 8
Move tile 5 down
    . 1 7
    5 6 2
    4 3 8
Move tile 1 left
    1 . 7
    5 6 2
    4 3 8
Move tile 7 left
    1 7 .
    5 6 2
    4 3 8
Move tile 2 up
    1 7 2
    5 6 .
    4 3 8
Move tile 6 right
    1 7 2
    5 . 6
    4 3 8
Move tile 3 up
    1 7 2
    5 3 6
```