## Search Exercise 6

# The 8-Puzzle

The 8-Puzzle is a simplified version of the better known

[15-Puzzle](https://en.wikipedia.org/wiki/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](https://murhafsousli.github.io/8puzzle/#/) and an interactive

8-Puzzle search algorithm demo can be found [here](https://deniz.co/8-puzzle-solver/).

(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

### 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:

## A couple of possible goal states:

NORMAL\_GOAL = [ [1,2,3],

[4,5,6],

[7,8,0] ]

SPIRAL\_GOAL = [ [1,2,3],

[8,0,4],

[7,6,5] ]

### Declarations of some example states

LAYOUT\_0 = [ [1,5,2],

[0,4,3],

[7,8,6] ]

LAYOUT\_1 = [ [5,1,7],

[2,4,8],

[6,3,0] ]

LAYOUT\_2 = [ [2,6,3],

[4,0,5],

[1,8,7] ]

LAYOUT\_3 = [ [7,2,5],

[4,8,1],

[3,0,6] ]

LAYOUT\_4 = [ [8,6,7],

[2,5,4],

[3,0,1] ]

### 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).<br>

\* 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)

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/crazy8heuristics.py

from bbmodcache.crazy8heuristics import bb\_misplaced\_tiles, bb\_manhattan

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)

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', 1000000, cost=cost, loop\_check=True, show\_state\_path=True)

#search(EP, 'BF/FIFO', 1000000, loop\_check=True, show\_state\_path=True)

## What else can we try? How about A\* ?

Probably you found that using a cost function was a bit of a disapointment.

If the cost is the same for every action, the effect will be the same as running a simple breadth-first search.

However, there is one more thing we should try. The \_\_A\*\_\_ algorithm uses both a cost function and a heuristic to estimate the total cost of a path running through a node and gives priority to states with the lowest sum of both the cost and the heuristic.

You should try that. And by now you should have seen enough examples of using the `search` function to write the code for doing \_\_A\*\_\_. All you need to do is specify functions for both the `cost` option and the `heuristic` option. Go to it:

## Specify and execute an A\* search run here:

Assuming that worked, you should perhaps compare the effectiveness of the two different heuristics: \_misplaced tiles\_ and \_Manhattan distance\_.

## Designing a systematic search algorithm testing experiment

In order to give a general analysis of how different search algorithms perform when applied to a problem, it is essential idea to have a clear and concise record of the tests that will be or were carried out during the investigation.

If the tests are carried out within a software system, an easy way to do this is to define the whole sequence of tests within a single file or within a function defination. Or, since I am using a Notebook file, I can simply specify sequence of test runs of the `search` function together within a code cell, as in the following cell.

Note that I have set the `return\_info` option to `True` and am catching the information resturned by assigning the results to variables (`T1` and `T2`).

And I have also set `dots=False`, which turns of the display of dots indicating progress of the search. You may not want to see the dots every time you run a given test.

prob1 = EightPuzzle( LAYOUT\_1, NORMAL\_GOAL )

T1 = search(prob1, 'BF/FIFO', 10000, loop\_check=False, dots=False, return\_info=True)

T2 = search(prob1, 'BF/FIFO', 1000000, loop\_check=True, dots=False, return\_info=True)

## Examining and Presenting Results

The result returned by `search` when `return\_info` is set to `True` contains a lot of information about the test that was run, the outcome obtained, the time taken and the size of the search space explored in terms of numbers of notdes generated, tested, left in the queue (and also those 'discarded' if the `loop\_check` is set to `True`).

Let us look at the info returned from test `T2`. (The `display` function is an inbuilt notebook function that is intended to present data objects in a way that is easily comprehensible by software developers.) The value of `T2` is as follows:

display(T2)

### Comparing results by generating a table

The results in the for displayed in the previous cell are not very easy to interpret and certainly do not facilitate comparison between different types of search.

To do that we may want to compile a table that presents key information in a clear and systemmatic way. The following cell contains a simple example of how we can do this by looping through a list of test results and printing out certain values taken from the search results:

TEST\_RESULTS =[T1, T2]

short\_tc = {"GOAL\_STATE\_FOUND" : "Y",

"NODE\_LIMIT\_EXCEEDED" : "!",

"SEARH-SPACE\_EXHAUSTED": "x"}

print("\n \*\*TESTS SUMMARY\*\*\n")

print("Test #max Result #gen #inQ Time s")

for i, test in enumerate(TEST\_RESULTS):

max = test['args']['max\_nodes']

tc = test['result']['termination\_condition']

stc = short\_tc[tc]

ng = test['search\_stats']['nodes\_generated']

nq = test['search\_stats']['nodes\_left\_in\_queue']

time = round( test['search\_stats']['time\_taken'], 2 )

print( f"{i:>3}: {max:>8} {stc} {ng:>8} {nq:>8} {time} ")

Not bad! But maybe should show solution path lengths as well?