CSDS-391 PA 1

Jonathan Yoo

March 2, 2023

1 Code Design

In order to properly organize the required functions, my functions are all contained within a python class called ‘ProgrammingAssignment1‘. I decided to store all of the puzzle states as a nested class ‘Puzzle‘.

```
class ProgrammingAssignment1:
    """
    Puzzle class stores all the appropriate functions for
    Programming Assignment 1, as well as initializes space for
    the Puzzle state and the max nodes to find the solution
    """

    def __init__(self, start_state):
        self.start_puzzle = self.Puzzle(start_state, start_state.index(0))
        self.max_nodes = None
        self.goal_puzzle = self.Puzzle([0, 1, 2, 3, 4, 5, 6, 7, 8], 0)

    class Puzzle:
        def __init__(self, state, blank):
            self.state = state
            self.blank = blank
```

The class ‘ProgrammingAssignment1‘ has three instance fields, ‘start_puzzle‘, which stores a Puzzle class representing the start state, ‘max_nodes‘, which stores the max nodes to be generated in a search, and ‘goal_puzzle‘, which stores a Puzzle class representing the goal state. The Puzzle nested class stores the state as an array of numbers 0-8, 0 representing the blank space, and a integer value ‘blank‘ that represents the location of the blank state. While this is less space efficient, I chose to store the puzzle states in this way in order to reduce the time and complexity of the ‘move()‘ function later on.

In addition to a nested class ‘Puzzle‘, I also created a nested class ‘Node‘ to use inside the search algorithms.

```
class Node:
        def __init__(self, value, path):
            self.value = value
            self.path = path
            self.cost = len(self.path)

        def __str__(self):
            return f'{self.value.state}'
```

The ‘value‘ field is meant to store a ‘Puzzle‘ class instance representing a state, path stores an array representing the path to get to the node, and the cost is the amount of moves from the starting state to get to the instance state. The ‘str()‘ overridden function is a string representation of the state array, for use in boolean operations and general debugging.

Several helper functions were also created to assist in general puzzle functions. These include:

```
expand(node) -> list of child nodes
list_valid_move(puzzle) -> list of valid moves
h_1(node) -> "h1" heuristic value
h_2(node) -> "h2" heuristic value
```

Each functions' return value is listed.

2 Code Correctness

The code runs properly as according to the to the 'command_file.txt' file. Running the 'pa_1.py' file should run whatever you put into the 'command_file.txt' file, and output proper documentation of each algorithm.

However, although both A* search algorithms run perfectly fine, beam search was never able to find a solution even with maxNode values ¿ 1000000. My explanation for this was that my algorithm did not have a reached hashmap, nor did it account for the cost it took to get to the state, which means that it potentially could infinitely loop through already reached states. I only used the h2 heuristic as an evaluation. The assumption I made was that I should not create a reached hashmap, and I would not factor in the state cost in evaluation.

3 Experiments

3.1 How does fraction of solvable puzzles from random initial states vary with the maxNodes limit?

Max Nodes(x1000)	A* h1	A* h2	Beam
1	30	10	10
2	40	20	10
3	30	30	10
4	40	40	10
5	30	20	10
6	30	90	10
7	60	120	10
8	40	130	10
9	50	180	10
10	40	30	10
11	50	230	10
12	50	70	10
13	40	40	10
14	40	70	10
15	70	70	10
16	60	50	10
17	50	240	10
18	40	240	10
19	50	80	10
20	60	250	10

Table 1: The data in all columns other than Max Nodes refers to the Maximum number of random moves the search algorithm was able to handle. Also note that due to extreme time constraints, after 30000 max nodes, the increments of the random moves for h2 A* had to be changed to increments of 100

Generally, as you increase the maxNodes limit, you can solve puzzles of a higher random initial state move count.

3.2 For A* search, which heuristic is better, i.e., generates lower number of nodes?

For A* search, as evidenced by the command_file.txt file and the tests from above, h2, the Manhattan Distance heuristic, is much better and consistently generates much greater nodes than h1, misplaced tiles. From the section 3.1 data, the max number of random moves generally means greater moves required to solve the puzzle, which means the depth of the solution is greater. This means that the higher number it is able to handle, the less nodes the algorithm generates.

3.3 How does the solution length (number of moves needed to reach the goal state) vary across the three search methods?

For both A* methods, they were always the same, and the most optimal path to solve the puzzle. For the beam search method, because my code was either wrong or that beam search was inherently very terrible as a search method, I was unable to get a large enough dataset to determine whether it would differ at higher move lengths. But for initial states where the optimal solution had very low cost,

3.4 For each of the three search methods, what fraction of your generated problems were solvable?

For h1 A*, with a sufficient enough maxNodes, all were solveable until about 100 random moves. For h2 A*, all were solveable, since my generated states were all solvable since they were generated through random moves. For beam, it could not solve past 10 random moves.

4 Discussion

4.1 Based on your experiments, which search algorithm is better suited for this problem? Which one finds shorter solutions (solutions with less number of moves)? Which algorithm seems superior in terms of time and space?

The A* search using the h2 heuristic is the best suited for this problem. A* h1 and A* h2 were equal in finding the shortest solutions. In terms of time and space, A* h2 was the best across the board. Also note however that the implementation of beam search I used did not include a reached array, which means that it could potentially loop through already reached states.

4.2 Discuss any other observations you made, such as the difficulty of implementing and testing each of these algorithms.

Since I have never coded this kind of a problem before, it was initially very challenging to code. But as I started coding, it was much easier to simply take the concepts from the textbook and implement them exactly to the tee through a python representation. Testing was also challenging, since each test of the search algorithm took very long times at higher maxNode values.