Name: Harris Shepard
SID: 862132345
Date: 05/01/2022
Project 1: 8 Puzzle Search
Report

Code Explanation

How to run code (commands):

```
PS C:\Users\harri\Documents\GitHub\cs170> python --version
Python 3.6.4
```

```
PS C:\Users\harri\Documents\GitHub\cs170> python "cs170 project1.py"
```

Data structures and data members:

Puzzle

- puzzle, a 1d array representing the puzzle, None represents the empty tile
- Num_moves, number of moves to get to the puzzle, starts at 0
- Cost, cost to get to this puzzle state, starts at 0
- Parent, a pointer to another Puzzle object, default is None
- Num_rows, number of rows

Problem

- Unexplored, a list of unexplored puzzles, starts with the initial puzzle
- Explored, a list of explored puzzles, starts empty
- Solution, the solution to the puzzle in list form
- Size, the number of rows/cols in the puzzles
- Found_solution, a Puzzle object that can be traced back to the solution

FAQ

Why not Priority Queue instead of lists for unexplored and explored?
Should use priority queue instead, saves time. My current implementation finds the next unexplored node by manual search.

Why use ‘None’ to represent the empty tile?

It makes searching through the puzzles more obvious. Python offers syntactic sugar when ‘None’ so it looks a bit better. If the empty tile were an asterisk, nothing would change much

Why 3 separate functions for each of the searches?
All 3 functions are very similar. I wanted to make the searches more extendable and flexible so that you could change 1 search and not affect the others.

Why the upper limit for the searches?

I wanted the searches to complete in a reasonable amount of time, currently that means a few seconds. If presented with an impossible puzzle, the program would exit after a few seconds and not run forever.

What are the Bugs?

Mainly the search is incomplete for adding explored nodes. **Once my program marks a puzzle state as explored, it would not be able to expand any states that also lead to that state even with a lower cost**.

I do not think it mattered for the heuristics given as they are pretty good at finding the right solution, but this should be noted.

The solution to this is to add another ‘if’ statement to check the ‘number of moves’ of the explored state and compare it to whether you could improve it and add new children with the improved cost to the unexplored state. I could not finish the solution in time

Ideally, you would be able to expand the node again and generate children of the Puzzle object using a new lowest cost

There are many bugs with the user input part of the project. That bit was rushed.

Analysis

		trivial			doable			oh_boy	
	UCS	Misplaced	Euclidean	UCS	Misplaced	Euclidean	UCS	Misplaced	Euclidean
depth	0	0	0	4	4	4	18	22	22
explored	0	1	1	17	7	7	30000	30000	30000
unexplored	1	0	0	14	0	0	17519	13312	7705
total	1	1	1	31	7	7	47519	43312	37705

length									

Explored Limit
Limit of 30000 simplified runtime at the cost of not letting the algorithm finish. Solutions were found before 30000 except in uniform cost search. Using an upper limit of 30,000 which takes about 13 minutes for all 3 searches

Uniform Cost Search
With a cost function of 1 per explored node, uniform cost search acted like breadth first search. It is very inferior to both the other A* search algorithms. In the ‘oh boy’ puzzle, it was not able to find a solution and only reached depth ~18 in 30000 nodes.

Misplaced vs Euclidean A* Search
The euclidean A* search algorithm performed the best time wise. You can see the unexplored for euclidean is lower as it reaches higher depths quicker than the misplaced a* search. Both the misplaced and euclidean searches should reach a similar total length because the depth of the solution is the same.

The Best Algorithm
If you were looking for a solution and not the best solution, euclidean would beat misplaced a* search.

Solution Depth
All 3 methods were able to find the solution with the exclusion of UCS not working in the “oh boy” puzzle exploring 30,000 nodes.

In all 3 methods, the first solution reached was the best solution. For uniform cost function this is by definition.

For the others, this indicates that the heuristic functions of the misplaced a* and euclidean a* were good heuristic functions. Otherwise we would see UCS with the correct solution initially and the a* searches with multiple solutions overwriting each other in the output

The Hardest Puzzle Anecdote
In another test, euclidean a* search reached the furthest depth before failing in the hardest problem possible for the 8 puzzle search.

Opinions

Multiple Solutions in A* search:
In practice I found that going for multiple solutions in A* search while correct in finding the best solution, I don't believe was necessary due to the accuracy of the heuristic functions.

As the heuristic functions led to the solution rather quickly, the program was taxed with having to expand other nodes to find a better solution. The expansion is very time consuming.

Conclusion

Time Allocation
Most of the work was implementing the data structures needed to implement a breadth first search. The cost functions were secondary and only required that I add a new field to the Puzzle class.

The multiple solutions of A* search required a small modification but was difficult to implement due to how my code was structured.

Evolution of code
Initially the code was using hashtables to store pairs of the puzzle array and the cost of the array. However, as I needed puzzles to store the number of moves as well it made sense to implement another class which let me print them much easier!

I had the misconception of needing to replace nodes in the explored tree to update the cost. The reality was that any explored node was already the optimal solution because the problem space is really simple and the heuristics given are good heuristic functions. In this scenario, it doesn't matter. In more complicated problems, you would have multiple paths to the same state that you would consider.

Optimization
My code is not very optimized, there are many different if statement checks and unnecessary variables declared for the sake of clarity. Python allows multiple variable declarations in 1 line as a tuple which could have reduced the time of the program.

Obviously a priority queue makes searching for minimum cost nodes trivial and should've been implemented.