Project 1: 8 Puzzle with 3 Different Algorithms

Hung Le, 862063377, hle026@ucr.edu, 11/1/2022
https://github.com/hungle132/cs170project1

Introduction:

In this project, I implemented 3 different search algorithms for the 8-puzzle that will search the possible states of the puzzle and find a solution to the puzzle. The puzzle is set up on a 3x3 board where the blank tile is represented by 0 in the puzzle. These 3 searches that are used in the project are uniformed cost search, A* search with a misplaced tile heuristic, and A* search with the use of the manhattan distance. The time in which these 3 algorithms take varies between each other and their space complexity varies between each other as well. I kept track of the depth at which the tree went and how many nodes were in the queue at a given time. I also kept track of the nodes expanded as well.

Uniform Cost Search:

In this search, every node will have been coded with a g(n) value of 0. As such, this means that every f(n) value will also have a value of 0. This shows that there is no weight to the expansion and it can keep expanding in any direction unless that state already exists inside a repeated state. The space of the search matches closely to the other two when the puzzle is much simpler to solve. However, as the puzzles have a deeper depth to them and more nodes are possible; the search could potentially take a long time due to the fact that we are checking every node possible. The time that it takes from the other two search methods will grow exponentially as we make the problem more complex. Furthermore, the search as it is now will sometimes solve the puzzle at depths 10 or above. The reason for this is because at some point we will run out of memory to even do more searches. Eventually, at the end of the search, we will be given out a goal puzzle with the number of nodes that were in the queue and the depth at which the search found it. If the search fails, then the program will output that there's no solution, or another way the search fails is by segmentation faulting due to the limitation on the amount of memory.

Misplaced Tile Heuristic:

In this search, the algorithm will search for the tiles that are misplaced in the heuristic and return an integer value that equates to the number of tiles moves needed to move the incorrect tiles into the right spot. Then the heuristic of the nodes will be changed to the integer that came back from the function that searches for the misplaced tiles. The f(n) of the nodes will change accordingly and will equal the heuristic value plus the g(n) value. This way we can organize the priority queue in ascending order with the lowest f(n) value. This way we can expand out the node that has the lowest f(n) value and find our solution according to how many tiles were misplaced at the start of the puzzle. This is more efficient than uniformed cost search as it is not expanding any nodes that have a higher f(n) value and don’t need to waste memory or time to work on those nodes. This way we can find the solution much faster than a uniformed cost search and at deeper depths as well.

Manhattan Distance Heuristic:

In this search, we are using the manhattan distance heuristic for the f(n) value in order to determine the number of moves the puzzle can be solved in. The formula that is used in this function is | r1 - r2 | + | c1 - c2 |. So when we calculate this, we’re able to get the tiles in their correct spots in a certain amount of moves. This way the priority queue can be organized with the f(n) moves according to the manhattan distance. We are able to find a solution very fast with this method and is the fastest-running algorithm out of the 3 used in this project. The power of this algorithm shows itself at deeper depths as it will run a lot faster than a uniformed cost search and show a difference with misplaced tiles.

Puzzle Data:

1 2 3
4 5 6
0 7 8

Uniformed Cost Search

depth	**Frontier max/curr**	**time**	**expanded**
2	4/3	Less than a sec	3

Misplace Tile Heuristic

depth	**Frontier max/curr**	**time**	**expanded**
2	3/2	Less than a sec	2

Manhattan Distance Heuristic

depth	**Frontier max/curr**	**time**	**expanded**
2	3/2	Less than a sec	2

1 2 3
5 0 6
4 7 8

Uniformed Cost Search

depth	**Frontier max/curr**	**time**	**expanded**
4	12	Less than a sec	23

Misplace Tile Heuristic

depth	**Frontier max/curr**	**time**	**expanded**
4	6/5	Less than a sec	4

Manhattan Distance Heuristic

depth	**Frontier max/curr**	**time**	**expanded**
4	6/5	Less than a sec	4

1 3 6
5 0 2
4 7 8

Uniformed Cost Search

depth	**Frontier max/curr**	**time**	**expanded**
8	157/156	.04	247

Misplace Tile Heuristic

depth	**Frontier max/curr**	**time**	**expanded**
8	16/15	.001	19

Manhattan Distance Heuristic

depth	**Frontier max/curr**	**time**	**expanded**
8	12/11	.001	12

1 3 6
5 0 7
4 8 2

Uniformed Cost Search

depth	Frontier max/curr	time	expanded
12	1128/1127	6.7	1885

Misplace Tile Heuristic

depth	Frontier max/curr	time	expanded
12	83/82	.01	131

Manhattan Distance Heuristic

depth	Frontier max/curr	time	expanded
12	28/27	.002	36

1 6 7
5 0 3
4 8 2

Uniformed Cost Search

depth	Frontier max/curr	time	expanded
N/A	N/A	Until mem runs out	N/A

Misplace Tile Heuristic

depth	Frontier max/curr	time	expanded
16	411/410	.6	701

Manhattan Distance Heuristic

depth	Frontier max/curr	time	expanded
18	151/150	.04	237

Conclusion:

As we can see with the results of the searches, the puzzles with depths less than 10 will have a similar search time from each other. Uniformed Cost Search is shown to take more time as the depth increases as compared to the other search. The space of each algorithm grew as we used puzzles with deeper depths. The searches with heuristics will prove to use less space and time than uniformed cost searches as they are more efficient with the way they find the goal puzzle. A bug that exists is that there is currently no false puzzle check at the beginning.