Introduction

The eight puzzle problem is a 3x3 board with one through eight displayed and one empty space. The original order of the numbers on the board are randomized and the goal of the puzzle is to move them around until they are the target configuration. In our case, it was 0 (or blank space), 1, 2 on the first row, 3, 4, 5 on the second row, and 6,7, 8 on the third and last row. You can move right, up, down, or left.

Figure 1. 8-puzzle problem goal configuration

Code Development

To solve this problem to reach the goal state, you must first create the problem. A program named random-board.py was created to make the randomized board in the same 3x3 format. A text file with the goal in the correct format was read using standard input and moved around randomly with the number of moves desired indicated. To solve the problem, a program was written in python employing the A* search. A* search is an informed search strategy which means it is able to use problem-specific knowledge to find the target configuration. The program consisted of a PriorityQueue class to act as the “frontier” which sorts the queue in decreasing order of desirability. A node class was created and used to keep track of each node information that was visited. A set class was used for the closed list aspect of the A* search. The closed list keeps track of what nodes were already visited, so there are no repeats. In our case, our state class was also able to move the node up, down, left, right, and if another state was equal to itself. The current board was saved as a state() and pushed onto the frontier priorityqueue(). The priorityQueue kept track of the nodes that were a possible next step in the search. The program implemented a while loop to pop the most cost efficient option from the frontier, saving that to the closed list, and expanding the chosen node. If the state was not the desired state, the program implemented the expand() function to find the surrounding nodes and adding them to the frontier with their calculated heuristic cost. Four heuristics were implemented to estimate the cost to the goal state. Heuristic 0 returned 0 for the cost for the most optimistic cost. Heuristic 1 was the Misplaced heuristic which calculated the amount of misplaced tiles from the goal tiles. Heuristic 2 was the Manhattan Distance which calculated the sum of the absolute difference between the goal position and the actual position. Heuristic 3 was the max heuristic which calculated maximum distance from the goal to any tile. Once the goal state was reached, d, N, V, and b were calculated. D is the depth of the optimal solution which was represented by the moves made to reach the final state. N is the maximum number of nodes stored in memory which was calculated by adding the amount of states and the length of the closed list. V is the total number of nodes visited or expanded which was calculated by

adding 1 after a node was chosen for expansion and added to the visited list. B is the approximate effective branching factor where N = b^d.

Experimentation

During the creation of the project, we tested various heuristics and beginning puzzle arrangements to evaluate the performance of the system. These tests sought to compare the effectiveness of the implemented heuristics and comprehend how the choice of heuristic affects the search process. These experiments revealed the Manhattan distance was the most efficient. It had the least amount of nodes expanded, the maximum number of nodes stored in memory, and the approximate effective branching factor.

Performance & Analysis

The analysis revealed the Manhattan distance was the most efficient while the max heuristic was the least. Overall, h(2) was the most cost effective in every aspect including V, N, D, and b as displayed below.

V =

	Minimum	Median	Mean	Maximum	Standard Dev.
H(0)	32	15185	56767.8	413825	100868.16
H(1)	13	1943	8429.66	20748	14167.78
H(2)	14	521	1310.2	8012	2498.24
H(3)	690	88721	108741	177442	47126.89

N=

	Minimum	Median	Mean	Maximum	Standard Dev.
H(0)	30	7981	20589.3	69540	25671.4
H(1)	12	1447	5429.35	25748	7167.78
H(2)	17	328	910.2	6841	1598.24
H(3)	42	88721	108741.2	342563	47126.89

D =

	Minimum	Median	Mean	Maximum	Standard Dev.
H(0)	7	17	16.78	24	5.41
H(1)	5	16	15.24	28	5.63
H(2)	6	15	14.81	23	5.71
H(3)	10	19	18.69	26	5.8

B =

	Minimum	Median	Mean	Maximum	Standard Dev.

H(0)	1.57	1.71	1.87	2.03	0.12
H(1)	1.42	1.61	1.59	1.67	0.061
H(2)	1.35	1.45	1.44	1.58	0.05
H(3)	1.77	1.65	1.69	1.98	47126.89

Limitations

The 8-puzzle problem was successfully resolved by our code utilizing the A* search technique and many heuristics. However, there were a few drawbacks:
Memory Usage: For complex problems, storing all examined states might result in substantial memory usage. A potential enhancement is to implement more memory-efficient data structures.
The admissibility or consistency of our heuristics for all puzzle configurations is not guaranteed, despite the fact that they produce accurate estimates in most cases. Better performance might be achieved with more research into advanced heuristics.

Conclusion

The 8-puzzle problem was successfully implemented using the A* search with a sequential set of board configurations leading back to the goal configuration. Four heuristics were created and tested against the randomized configuration to reach the goal. The sum of Manhattan distances revealed to be the most efficient in the search with a cost effective d, N, V, and b. We also talked about our approach's weaknesses and possible upgrades, highlighting the opportunities for future study in this area.

Cited

Nik. (2022, February 23). *Calculate Manhattan distance in Python (City Block Distance) • datagy*. datagy. https://datagy.io/manhattan-distance-python/

Swift, N. (2020, May 29). *Easy A* (star) pathfinding*. Medium. https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2

Sonawane, A. (2020, June 24). *Solving 8-puzzle using a* algorithm.* Medium. https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288