8 – Puzzle and Search Algorithms

Overview:

In this project I created an 8-puzzle program in C++, generating random solvable* 8-puzzle boards along with various search algorithms to solve said puzzle. The program utilizes Depth-First Search (DFS), Best-First Search (BestFS), and A* Search algorithms. Two different heuristics are applied to estimate the distance between a given board state and the goal state:

(i) The number of misplaced tiles
(ii) The Manhattan Distance – sum of all tiles 'distance' to their goal state

The performance of each algorithm is analyzed by tracking run times for time complexity, and priority queue/stack/visited list size for space complexity. It should be noted that since DFS performs significantly worse than A* and BestFS, which show comparable performance's, DFS was left out in some of the testing.

Algorithms:

Depth-First Search (DFS):

- Explores as deep as possible before backtracking, using a stack for storing the nodes(boards) to be visited.
- Explores child boards by trying to slide empty space up, then right, then down, and finally left depending on whether the tile can be slid that way and if doing so results in a board which hasn't already been visited.

Best-First Search (BestFS):

- Explores nodes based on which board in queue has the lowest heuristic (misplaced tiles or Manhattan distance)
- Uses a priority queue to store the to be visited boards, evaluating states based on the current heuristic function (min. heap - lowest heuristic is next in queue) and tie-breaker value (boards queued first will be visited before boards queued later if heuristic is same)

A-Star Search (A*):

- Does everything BestFS does while also keeping track of the distance a node is from the start.
- Thus, boards are placed in the priority queue not only based on the heuristic (h(n)) but also the path length (g(n) using the formula f(n) = g(n) + h(n), f(n) representing the total estimated cost.

Heuristics: *To switch between un-comment in one and re-comment out the other

- ***h1(n)*:** Number of misplaced tiles (the count of tiles not in their goal state).
- ***h2(n)*:** Manhattan Distance (sum of distances of each tile from its goal position).

Sample Solution/Output: *Using ASTAR on sample easy sample board

```
Path:
-------------
| 3 | 1 | 2 |
-------------
| 4 | 7 | 5 |
-------------
| 6 | 8 | 0 |
-------------

-------------
| 3 | 1 | 2 |
-------------
| 4 | 7 | 5 |
-------------
| 6 | 0 | 8 |
-------------

-------------
| 3 | 1 | 2 |
-------------
| 4 | 0 | 5 |
-------------
| 6 | 7 | 8 |
-------------

-------------
| 3 | 1 | 2 |
-------------
| 0 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------

-------------
| 0 | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
```

A* Search Stats			
Board Number	Queue Size	Visted List Size	Time(milliseconds)
0	1	1	0.0045
1	2	2	0.1385
2	3	3	0.1931
3	5	4	0.2716
4	6	5	0.3279

Table 1: A* search statistics, queue/visited list sizes and runtime, taken each time the algorithm visits a new board on easy puzzle board.

Heuristic Comparison: *Using ASTAR on medium sample board

Manhattan Distance

ASTAR Search Stats – Sample Board 1			
Board Number	Queue Size	Visted List Size	Time (milliseconds)
0	1	1	0.0225
1	3	2	0.1818
2	5	3	0.2574
2324	1337	2325	4126.262
2325	1337	2326	4129.26
2326	1337	2327	4133.063
4613	2679	4614	16927.48
4614	2681	4615	16937.39
4615	2682	4616	16948.63

Misplaced Tiles

ASTAR Search Stats – Sample Board 1			
Board Number	Queue Size	Visted List Size	Time (milliseconds)
0	1	1	0.0061
1	3	2	0.1234
2	3	3	0.1597
4998	2932	4999	24799.14
4999	2933	5000	24812.13
5000	2933	5001	24824.61
9997	5917	9998	150748.6
9998	5918	9999	150786.4
9999	5919	10000	150823.4

ASTAR Search Stats – Sample Board 2			
Board Number	Queue Size	Visted List Size	Time (milliseconds)
0	1	1	0.0049
1	3	2	0.1796
2	5	3	0.2557
73	53	74	7.7403
74	54	75	7.9321
75	54	76	8.0738
141	89	142	20.6602
142	91	143	20.9989
143	92	144	21.2671

ASTAR Search Stats – Sample Board 2			
Board Number	Queue Size	Visted List Size	Time (milliseconds)
0	1	1	0.0041
1	3	2	0.0914
2	5	3	0.144
231	156	232	54.7377
232	156	233	54.9934
233	156	234	55.2501
463	280	464	178.6613
464	280	465	179.2048
465	281	466	179.958

ASTAR Search Stats – Sample Board 3			
Board Number	Queue Size	Visted List Size	Time (milliseconds)
0	1	1	0.0041
1	4	2	0.1719
2	5	3	0.2233
740	462	741	440.4308
741	462	742	441.3455
742	464	743	442.7684
1424	899	1425	1637.04
1425	901	1426	1640.215
1426	902	1427	1642.666

ASTAR Search Stats – Sample Board 3			
Board Number	Queue Size	Visted List Size	Time (milliseconds)
0	1	1	0.0126
1	4	2	0.1107
2	5	3	0.1544
2603	1577	2604	4824.532
2604	1577	2605	4827.182
2605	1578	2606	4831.138
5427	3185	5428	20953.12
5428	3186	5429	20962.02
5429	3187	5430	20970.82

Table 2: A* search statistics using both Manhattan Distance and Misplaced Tiles Heuristics on 3 sample puzzle boards. For each run the first, middle and last 3 stats are reported. For Misplaced tiles on sample Board 1 we can see the Board Number reaches 10k, this means that the programs was terminate before reaching the goal board.

Conclusion:

The Manhattan Distance heuristic leads to a more efficient/faster search, with lower average runtimes, queue sizes, and visited lists compared to the misplaced tiles heuristic. In particular, as seen in Sample Board 1, using the misplaced tiles heuristic resulted in the search terminating before reaching the goal state was even reached due to the program exploring 10,000 boards, whereas Manhattan Distance consistently found the solution more efficiently. Overall, the Manhattan distances consistently gives a better description of how close a board is to the goal state when compared to Misplaced Tiles, resulting in an estimate of 3 to 4 times better space complexity and over 10 times better runtimes.

<u>Algorithm Comparison</u>: *failed if algorithm explores 10k boards before finding Goal

Run Time(milliseconds) - Algorithm Comparison – Misplaced Tiles									
Trial #	ASTAR			BestFS			DepthFS		
	Time(ms)	Outcome	Max Q	Time(ms)	Outcome	Max Q	Time(ms)	Outcome	Max Q
1	3089.426	Success	1685	362.7698	Success	562	43464.92	Failed	7802
2	944.5507	Success	948	297.0073	Success	539	43936.52	Failed	7787
3	39123.01	Failed	5756	61.981	Success	229	45014.33	Failed	7787
Avgs.	14385.7	2/3	2796.3	240.6	3/3	443.3	44138.6	0/3	7792.0

Table 3: Compares the run times, success rates, and max queue sizes, along with their averages, of A* Search, BestFS, and DepthFS on 3 randomly generated 8-puzzle boards, while using misplaced tiles heuristic.

Conclusion:

Based on these results comparing A*, BestFS, and DFS, we can see that DFS is very bad at least on this problem and when compared to A* and BestFS. In this randomly generated test suite, DFS failed all three trials due to exceeding the 10k explored board limit. This may be partially attributed to the use of the misplaced tiles heuristic, but either way we can conclude that DFS is not suited for this problem and will be removed from further testing to conserve time and resources. Now let's focus on A* and BestFS to see who's the champ of search algorithms.

<u>ASTAR vs BestFS</u>:

Run Time(milliseconds) - Algorithm Comparison – Misplaced Tiles						
Trial #	ASTAR			BestFS		
	Time(ms)	Outcome	Max Q	Time(ms)	Outcome	Max Q
1	1072.741	Success	921	99.4656	Success	266
2	93.0324	Success	278	58.5257	Success	228
3	133376.4	Failed	5696	317.6455	Success	249
4	3301.296	Success	822	165.776	Success	194
5	447.1628	Success	377	192.6658	Success	238
6	6955.688	Success	1398	32.0295	Success	67
7	6112.607	Success	1280	81.7043	Success	137
8	2765.115	Success	823	241.5326	Success	242
9	1292.792	Success	566	70.737	Success	101
10	4957.727	Success	1085	11.4877	Success	47
Avgs.	14385.7	90%	1324.6	240.6	100%	176.9

Table 4: Compares the run times and success of A* Search vs BestFS on 10 randomly generated 8-puzzle boards, while using Manhattan distance heuristic

Conclusion:

When comparing A* and BestFS using the Manhattan Distance heuristic, the results indicate that BestFS is significantly more efficient in terms of time and space complexity. BestFS consistently found solutions faster and with a smaller max queue size than A*, often by a significant amount. However, A* guarantees a solution path that is at least as short as BestFS, thus the choice between the two algorithms depends on how concerned we are about resources (memory and time) and how optimal we want our solution to be.

Setup:

In order to get everything set up so you can start finding some anagrams you will want to:

1. Get Python: make sure you have Python downloaded(version 3) - https://www.python.org/downloads/

2. Get an IDE: I recommend getting an IDE like visual studios to make running and navigating the code so much easier - https://code.visualstudio.com/download

3. Get Python Code: download the code, make sure the code you have matches the Python version you are running and that all the files are in the correct location.

4. Open/Run Code: Once you are all set up in your IDE and have the correct version of Python installed. Next, you will want to open the folder containing the Python code and run the EightPuzzle_Main.py file.

5. Follow Menu Instructions: From there you should be good to go, the terminal will display the starting board and prompt you for what search algorithm you want to use to solve it. It will then display a counter for every 1k boards it visits and if the goal board is found before we visit 10k boards, the path from start to goal will be displayed. If not, the program will terminate and the search was a failure. Statistics of performance will be printed to a CSV automatically.

Optionally, there are some parts of the code that you might want to play with, for example:

EightPuzzle_Main.py:

- If you want to further compare the algorithms simply comment out the main and un-comment lines 58 or 59 at the bottom.

```
57      #Run algorithm comparison comment path printer
58      #algComparer(3)
59      #BFSvsASTAR(10)
```

- If you want to adjust or get rid of the explored board limit either comment out these if statements at the bottom of each search function or adjust the number ‘10000’ on line 151

```
151             if nextTry >= 10000:
152                 print("ASTAR is bad... giving up")
153                 FOUT.write("Max Queue Size: %d\n" % maxQueueSize)
154                 print("ASTAR: Max Queue Size: %d\n" % maxQueueSize)
155                 FOUT.close()
156
157                 return foundSolution
```

BoardClass.py:

- If you want to solve specific boards go into the initializePuzzleBoard() function and add the board into the sample boards section following the others as a template. If you were to un-comment line 108 – 110 then you would get that board every time you initialize a puzzle board. X and Y correspond to the coordinates of the empty tile (0) when indexing at 0.

```
106             # Sample Boards
107             # Board 1
108             #self.Board = [[6, 2, 1], [3, 8, 5], [4, 0, 7]]
109             #self.X = 2
110             #self.Y = 1
111
112             # Board 2
113             #self.Board = [[6, 1, 2], [7, 8, 4], [3, 0, 5]]
114             #self.X = 2
115             #self.Y = 1
116
117             # Board 3
118             #self.Board = [[1, 7, 5], [6, 0, 2], [4, 3, 8]]
119             #self.X = 1
120             #self.Y = 1
```

- If you want to switch the heuristic you’re using simply comment out one of the functions and un-comment in the other (top function is Manhattan distance, bottom function is misplaced tiles)

```python
    #=====================================================
    def computeDistanceFromGoal(self) -> None:
        """Computes the Manhatten Distance from the Goal board,
        where Manhatten Distance = (sum of misplaced distances for all tiles)"""

        sum = 0

        for row in range(BoardClass.N):
            for col in range(BoardClass.N):
                curTile = self.Board[row][col]

                goalRow, goalCol = BoardClass.GoalTiles[curTile]
                sum += (abs(row - goalRow) + abs(col - goalCol))

        self.Heuristic = sum
    #=====================================================
    '''
    def computeDistanceFromGoal(self) -> None:
        """Computes the number of misplaced tiles from the Goal board"""
        numberMisplaced = 0

        for row in range(BoardClass.N):
            for col in range(BoardClass.N):
                if (self.Board[row][col] != BoardClass.GOAL[row][col]):
                    numberMisplaced += 1

        self.Heuristic = numberMisplaced
    '''
    #=====================================================
```

<u>References</u>:

This project was developed with the assistance of AI-powered tools:

- **ChatGPT** – Used for generating explanations, refining documentation, and structuring this report.
- **Visual Studio Code Copilot** – Assisted with code suggestions and optimizations during implementation.