Project 1 CS205: Introduction to Artificial Intelligence, Dr. Eamonn Keogh

Kaushal Bandaru
SID: 862393040
kband008@ucr.edu
15-May-2023

In completing this project I consulted:

- the blind and heuristic search from the slides.
- Python documentation of 3.x.
- the sample report along with the puzzles provided in the project handout.
- these links to get an idea of how the program flows and how it works: https://github.com/anchitab/cs170-project1 and https://github.com/DanialBeg/CS170-EightPuzzle/blob/main/eightpuzzle.py
- this link to write the definition of Manhattan Distance: https://www.geeksforgeeks.org/sum-manhattan-distances-pairs-points/
- the built-in modules used - matplotlib, time, and copy - to prepare graphs, to make a note of the time spent using a search method, and to make deep copies of my problem state.

All the important code is original apart from the points mentioned above.

The complete code can be found at: https://github.com/bkaushal07/cs205_project1

Outline of this report:

1 Introduction

The eight-tile puzzle is a classic, fun, challenging, and strategic game that requires critical thinking and problem-solving skills to solve it. The objective of this game is to slide tiles around a 3x3 grid until they are arranged in the correct order but there is a blank space instead of the digit 9. Initially, a 15-tile puzzle was introduced, which later evolved into 8-tile and 25-tile puzzles, where the former is the most common variant played today.
In this project, we were assigned to solve an eight-tile puzzle using three different search algorithms namely: 1) Uniform Cost Search, 2) A* with the Misplaced Tile Heuristic, and 3) A* with the Manhattan Distance Heuristic. I have used Python to write the program and implement these algorithms, and tried to follow the pseudo-code taught in class as close as possible. I have evaluated the program using the eight test cases given in the handout along with some of my own tests on all three algorithms. The blank space in my program is represented by an asterisk and while inputting a user's own test case 0 must be used for a blank space.

2 Comparison of Search Algorithms

As mentioned above, I compared three different algorithms to test the efficiency in terms of time and memory. We know that A* algorithm, f(n) = g(n) + h(n), where f(n)The evaluation can be found in the below section in more detail.

2.1 Uniform Cost Search

Uniform Cost Search (UCS) is an algorithm that takes into account the cost of each path and chooses the path with the lowest overall cost, where the cost comes only from g(n) since h(n) = 0. It is a complete algorithm, meaning that if a solution exists, it is guaranteed to find it. UCS is also optimal, meaning that it will find the shortest path possible. However, the time complexity and space complexity are pretty high i.e., $O(b^d)$ [1]. Hence, f(n) = g(n) in UCS, and we can consider this as our baseline model.

[1] From Prof. Keogh's Blind Search slides.

2.2 A* with Misplaced Tile Heuristic

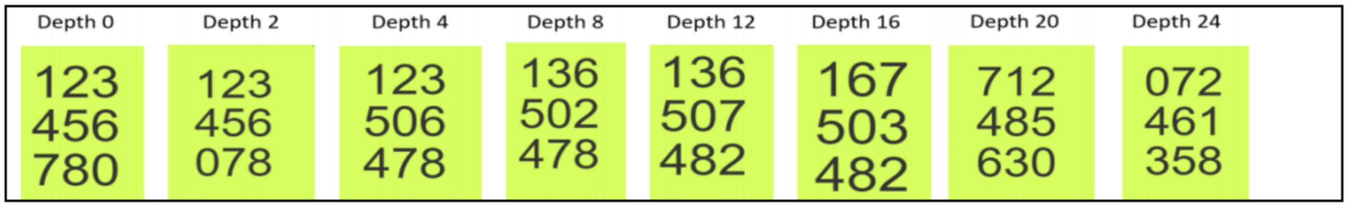
In this algorithm, each tile's placement in the puzzle board is taken into account, and its position is compared to the desired position. The heuristic function estimates the distance remaining for the goal state to be reached. It is calculated by counting the number of tiles that are currently misplaced in comparison to the goal state. The algorithm considers both the cost of reaching the current state and the estimated cost of reaching the goal state, where the latter is determined by the misplaced tile heuristic function. It then selects the path with the lowest total cost. This approach is known to be both admissible and consistent, providing an optimal solution with the least possible cost while avoiding the unnecessary exploration of bad paths. Hence, f(n) = g(n) + h(n), where g(n) is the cost from current to initial state, and h(n) is the number of the misplaced tiles between current and goal state.

2.3 A* with Manhattan Distance Heurisitc

In this algorithm, the heuristic function estimates the distance remaining for the goal state to be reached. The Manhattan distance heuristic calculates the distance between each tile's current position and its desired position in the goal state, then sums up these distances for all tiles in the puzzle. It measures the total distance each tile has to move vertically and horizontally to reach its desired position. The algorithm takes into account both the cost of reaching the current state and the estimated cost of reaching the goal state, where the latter is determined by the Manhattan distance heuristic function. It then chooses the path with the lowest total cost. This approach is also known to be both admissible and consistent, providing an optimal solution with the least possible cost while reducing the unnecessary exploration of bad paths. Hence, f(n) = g(n) + h(n), where g(n) is the cost from current to initial state, and h(n) is the manhattan distance between current and goal state.

3 Evaluation

The evaluation was performed on the eight test cases of eight different depths which was mentioned in the project handout. Figure 1 shows those eight test cases

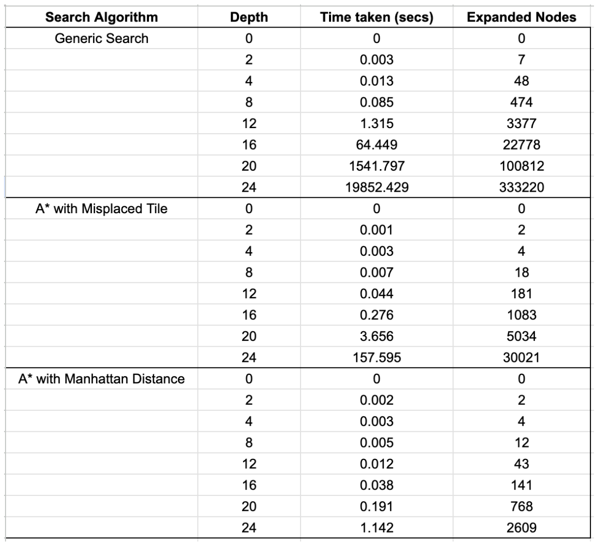


Depth 0	Depth 2	Depth 4	Depth 8	Depth 12	Depth 16	Depth 20	Depth 24
123	123	123	136	136	167	712	072
456	456	506	502	507	503	485	461
780	078	478	478	482	482	630	358

Figure 1: Evaluation done on eight test cases

The results can be seen in figure 2, where all the metrics were collected on the test cases that were given in the project handout. As shown in the metrics figure, uniform cost search performed very poorly when the depth of the puzzle increased. After a certain amount of rows, we can also observe that the number of expanded nodes (memory) is increased drastically. Whereas in the heuristic search algorithms, both perform fairly well with respect to the time and space complexity, although A* with misplaced tile takes more time and space when the depth increases (just like UCS).

By further looking at the graph shown in Figure 2, we can see that Manhattan distance is almost like a straight line (almost parallel to the x-axis), which means that it is very quick.



Search Algorithm	Depth	Time taken (secs)	Expanded Nodes
Generic Search	0	0	0
	2	0.003	7
	4	0.013	48
	8	0.085	474
	12	1.315	3377
	16	64.449	22778
	20	1541.797	100812
	24	19852.429	333220
A* with Misplaced Tile	0	0	0
	2	0.001	2
	4	0.003	4
	8	0.007	18
	12	0.044	181
	16	0.276	1083
	20	3.656	5034
	24	157.595	30021
A* with Manhattan Distance	0	0	0
	2	0.002	2
	4	0.003	4
	8	0.005	12
	12	0.012	43
	16	0.038	141
	20	0.191	768
	24	1.142	2609

Figure 2: Metrics of 3 algorithms

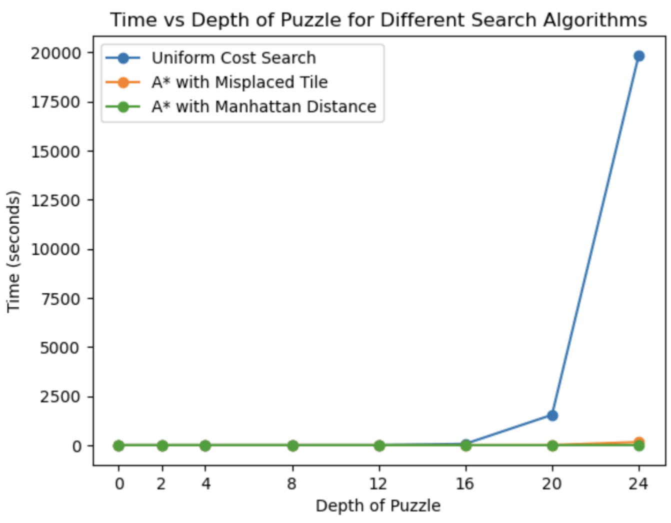


Figure 3: Time (secs) vs Depth

Additionally, looking at figure 4, we can also see that Manhattan distance takes far less space compared to the other two algorithms, which means that it is space efficient too.

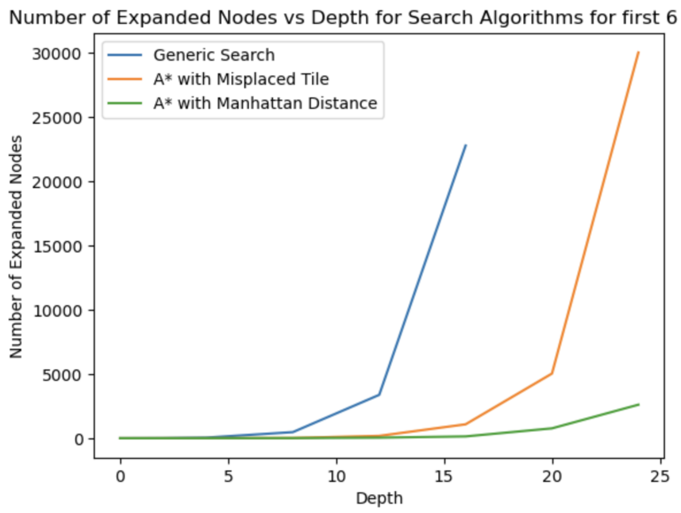
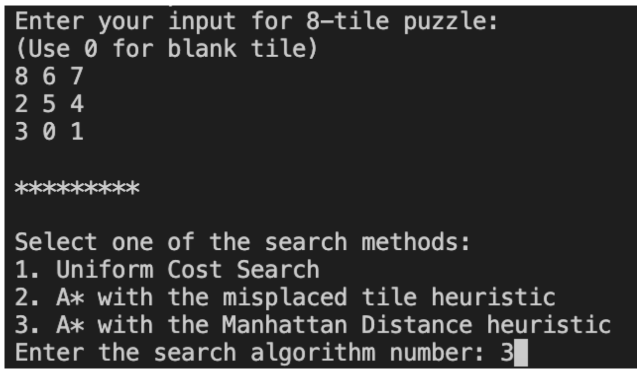


Figure 4: Expanded nodes vs Depth for first 6 test cases

3.1 Own Test Cases Evaluation

I also performed an evaluation on a puzzle of depth 31. Looking at the previous results, I only decided to test this puzzle with Manhattan Distance heuristic. The input of the puzzle is shown in figure 5, and the result of it is shown in figure 6.

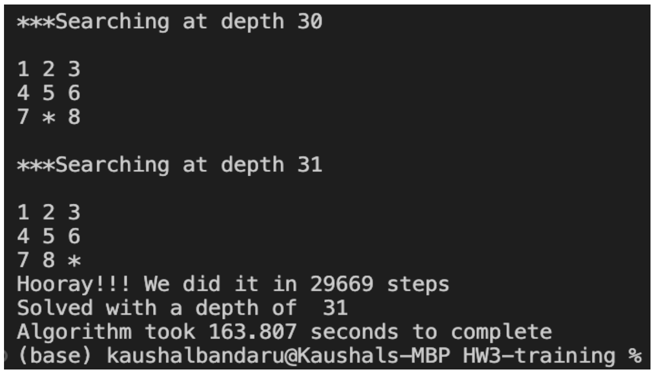


```
Enter your input for 8-tile puzzle:
(Use 0 for blank tile)
8 6 7
2 5 4
3 0 1

*********

Select one of the search methods:
1. Uniform Cost Search
2. A* with the misplaced tile heuristic
3. A* with the Manhattan Distance heuristic
Enter the search algorithm number: 3
```

Figure 5: Input to a puzzle of Depth 31



```
***Searching at depth 30

1 2 3
4 5 6
7 * 8

***Searching at depth 31

1 2 3
4 5 6
7 8 *
Hooray!!! We did it in 29669 steps
Solved with a depth of  31
Algorithm took 163.807 seconds to complete
(base) kaushalbandaru@Kaushals-MBP HW3-training %
```

Figure 6: Output of the puzzle of Depth 31

We can see that the puzzle of depth 31 was solved in approximately 29 thousand steps and in about 2.5 minutes (160 secs) with Manhattan Distance Heuristic.

4 Conclusion

In this project we performed an implementation of three search algorithms on the eight tile puzzle following the pseudo-code provided. From the results and the evaluation metrics obtained we can confidently conclude that the heuristic algorithms were faster and more memory efficient than the generic search algorithm. Also, between the heuristic algorithms, we could observe that A* with manhattan outperformed the misplaced tile heuristic.

Trace of the puzzle of depth 8 test case

The following is the trace of test case 4 using the Manhattan Distance algorithm:

```
Choose an option:
1. Use Default Puzzle
2. Enter your own 8-tile Puzzle
Enter the option: 1
Enter any puzzle number between 1 and 8 (1 easiest and 8 hardest): 4
You selected this puzzle:
1 3 6
5 * 2
4 7 8

*********

Select one of the search methods:
1. Uniform Cost Search
2. A* with the misplaced tile heuristic
3. A* with the Manhattan Distance heuristic
Enter the search algorithm number: 3

***Searching at depth 1

1 3 6
5 2 *
4 7 8

***Searching at depth 1

1 3 6
* 5 2
4 7 8

***Searching at depth 2

1 3 *
5 2 6
4 7 8

***Searching at depth 2

1 3 6
4 5 2
* 7 8

***Searching at depth 3

1 * 3
5 2 6
4 7 8

***Searching at depth 3
```

```
1 3 6
4 5 2
7 * 8

***Searching at depth 4

1 2 3
5 * 6
4 7 8

***Searching at depth 4

1 3 6
4 5 2
7 8 *

***Searching at depth 5

1 2 3
* 5 6
4 7 8

***Searching at depth 6

1 2 3
4 5 6
* 7 8

***Searching at depth 7

1 2 3
4 5 6
7 * 8

***Searching at depth 8

1 2 3
4 5 6
7 8 *
Hooray!!! We did it in 12 steps
Solved with a depth of  8
Algorithm took 0.005 seconds to complete
```

Main code (rest of the code can be found on Github):

https://github.com/bkaushal07/cs205_project1

eight_tile_puzzle.py

```
# Generic Search algorithm
def generic_search(root: Node, algorithm):
    nodes = [root] # nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
    max_depth = 0 # to track deepest level
    expanded_nodes = 0 # count of nodes that have been expanded or visited
    while nodes:
        node = nodes.pop(0) # popping the first element in the list
        if node.problem == goal_state: # check if at goal state
            print(f'Hooray!!! We solved it!! \nExpanded {expanded_nodes} nodes')
            return node
        nodes = queueing_function(node, nodes, algorithm) # nodes is updated with the newly generated nodes as
per chosen search algorithm
        expanded_nodes+=1
        max_depth = max(max_depth, node.depth)
    print('NADA!!')
    return None # return None if no solution


# Calculates the number of misplaced tiles in the given puzzle state compared to the goal state.
def heuristic_misplaced_tile(root: Node):
    misplaced_count = 0
    for i in range(len(root.problem)):
        for j in range(len(root.problem[i])):
            # Check if the tile at position (i, j) is misplaced
            if goal_state[i][j] != root.problem[i][j] and not (i == len(root.problem[i]) - 1 and j == len(root.problem) -
1):
                misplaced_count += 1
    return misplaced_count


# Calculates the Manhattan distance heuristic for the given puzzle state compared to the goal state.
def heuristic_manhattan_distance(root: Node):
    total_distance = 0
    for i in range(len(root.problem)):
        for j in range(len(root.problem[i])):
            if root.problem[i][j] != 0:
                # Find the target position (x, y) of the tile in the goal state
                for x in range(len(goal_state)):
                    if root.problem[i][j] in goal_state[x]:
                        y = goal_state[x].index(root.problem[i][j])
                        break
                # Calculate the Manhattan distance between the current position (i, j) and the target position (x, y)
                total_distance += abs(x - i) + abs(y - j)
    return total_distance
```