CS 205 : Artificial Intelligence
Project 1 : The Eight Puzzle, Dr. Eamonn Keogh

Anirudh Tulasi
SID : 862395278
Email : atula002@ucr.edu
Date : May 15th, 2023

In completing this project, I consulted:

- The Blind Search and Heuristic Search lecture slides and notes annotated from lecture.
- Python Documentation: (https://docs.python.org/3.10/contents.html)
- Data Structure Visualizations by USF (https://www.cs.usfca.edu/~galles/visualization/Algorithms.html)
- For the screenshots of generated puzzles: (Deniz Gürkaynak, 8-Puzzle Solver (https://deniz.co/8-puzzle-solver/)
- Pseudo code of general search from the handout and slides.

All important code is original. Unimportant subroutines that are not completely original are:

- All subroutines used from heapq, to handle the node structure of states.
- All subroutines used from numpy, to handle numerical computation efficiently.
- All subroutines used from copy, to deepcopy and correctly modify states.
- Part of general Search Algorithm in the search.py was adopted from the pseudo code provided in the handout and slides as required.

Code Structure :
Project ->

- eight_puzzle.py
- search.py
- node.py

Outline of the report:

CS 205 : Project 1 : The Eight Puzzle

Anirudh Tulasi | SID : 862395278 | May 15th, 2023

This puzzle problem is a basically a 3 x 3 tile solving puzzle which is a shorter version of the original 15 puzzle. According to Daniel Ratner and Manfred Warmuth *"The problem has been extended to an nxn board, and finding a shortest solution for the extended puzzle is NP-hard and computationally infeasible"* [1] . In our case for a 3x3 tile solving puzzle which is also called the eight puzzle, we have nine tile spaces and one empty tile meaning that a tile can only be moved up, down, left, or right depending on the availability of space on the tile's adjacent sides. One can use a desired state as the intial and goal states. However, the generalized version assumes that bottom right tile to be empty and other tiles to be in the order of one to eight as shown in Figure 1.

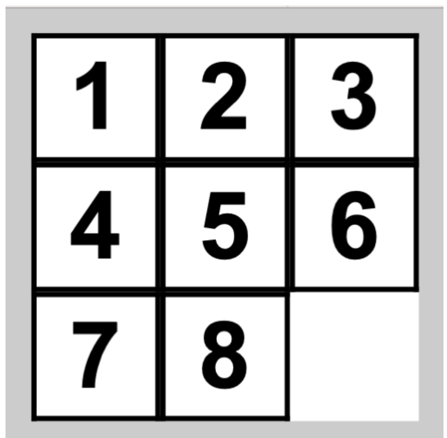

Figure 1 : A solved 8-Puzzle game [2]

In this project, we were asked by Dr.Keogh to write a program in any language that would solve this puzzle for which I have chosen Python 3.10. I have implemented the code in Python and compared the results for depth ranging from 0 to 31on UCS (Uniform Cost Search Algorithm), A * Misplaced Tile Algorithm and A* Manhattan Distance Algorithm. The code and the results for the same are attached in the report below. For the structure of the report, I have followed the sample report structure provided in the "*project handout*". [3]

Exploring Algorithms

For this project, I have implemented Uniform Cost Search, A* with Misplaced Tile heuristic, and A* with Manhattan Distance heuristic. The general search function pseudo code is already provided in the "*slides and the project handout*" [4] and was built on top of it.

Uniform Cost Search (UCS):
"UCS, as stated in the project brief, is essentially a variant of the A algorithm where h(n), the heuristic function, is fixed to zero"* [5]. Essentially, this approach prioritizes the node expansion with the lowest cost, with the cost being defined by g(n). For our project, there are no expansion

weights, meaning each node expansion has a cost of 1 which shows the real-world scenario where sliding a tile in any direction requires the same level of effort.

Misplaced Tile Heuristic:
The second approach that we used is the A* algorithm integrated with the Misplaced Tile Heuristic. This heuristic measures the number of tiles in the puzzle that are not in their correct position. For example, consider the states in figure 1 as the goal states. Now the initial states for a puzzle are as shown below in Figure 2.

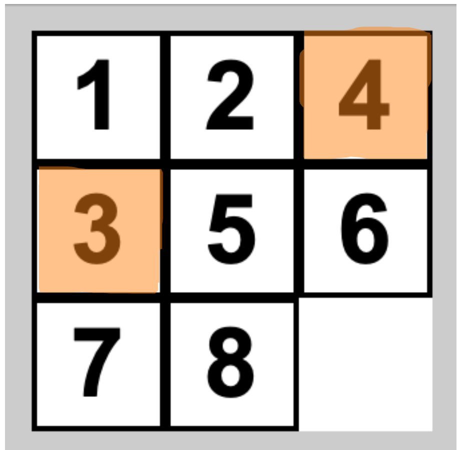

Figure 2 : A random 8-Puzzle game [6]

From the above image, we can see that the initial states are [1,2,4,3,5,6,7,8,0] and the goal state is [1,2,3,4,5,6,7,8,0] from the figure 1. Here, the tiles 4, 3 are not in the correct state when compared with the goal states. Taking this as heuristic, the program would expand further such that goal state is reached optimally.

Manhattan Distance Heuristic:
This follows a similar approach as that of the Misplaced tile heuristic but also calculates the moves required to reach it's goal state. For the example in figure 2, we can see that [3, 4] are misplaced tiles and each would require at least three moves to reach their goal state from the initial or current state.

This is much more optimal than the misplaced tile because it will also consider the cost to move from the initial or current state to the goal state.

Comparing the algorithms on the testcases:

In the handout, we were given a set of test cases for the depth-levels 0,2,4,8,12,16,20 and 24. These test cases were used to test the program that I've built and all the required performance metrics have been captured so that I can compare the working of these three algorithms on the different test cases and come to conclusions on what type of algorithm works best in the varied types of cases. I have included the data recorded in a tabular format as shown below. The table consists of six columns where the time taken is measured in milliseconds (ms). Nodes expanded column is total number of nodes expanded in the search space which gives us an idea of how my the search program had to go through to find a solution. Max Queue size is used to find the maximum number of nodes that were in a queue at any point of time. This is generally used to know about the memory usage of the search program.

Algorithm	Depth	Time taken in ms	Nodes Expanded *	Max Queue Size	Test Case
UCS	0	0.2 ms	0	1	1 2 3 4 5 6 7 8 0
ASTAR_MISPLACED	0	0.3 ms	0	1	1 2 3 4 5 6 7 8 0
ASTAR_MANHATTAN	0	0.5 ms	0	1	1 2 3 4 5 6 7 8 0
UCS	2	0.6 ms	3	4	1 2 3 4 5 6 0 7 8
ASTAR_MISPLACED	2	1.6 ms	2	3	1 2 3 4 5 6 0 7 8
ASTAR_MANHATTAN	2	3.8 ms	2	3	1 2 3 4 5 6 0 7 8
UCS	4	3.5 ms	36	32	1 2 3 5 0 6 4 7 8
ASTAR_MISPLACED	4	1.2 ms	4	6	1 2 3 5 0 6 4 7 8
ASTAR_MANHATTAN	4	3.0 ms	4	6	1 2 3 5 0 6 4 7 8
UCS	8	24.5 ms	285	170	1 3 6 5 0 2 4 7 8
ASTAR_MISPLACED	8	1.7 ms	19	16	1 3 6 5 0 2 4 7 8
ASTAR_MANHATTAN	8	6.3 ms	8	12	1 3 6 5 0 2 4 7 8
UCS	12	92.4 ms	1926	1142	1 3 6 5 0 7 4 8 2
ASTAR_MISPLACED	12	12.3 ms	131	83	1 3 6 5 0 7 4 8 2
ASTAR_MANHATTAN	12	14.9 ms	36	28	1 3 6 5 0 7 4 8 2
UCS	16	448.7 ms	13101	6562	1 6 7 5 0 3 4 8 2
ASTAR_MISPLACED	16	50.6 ms	701	411	1 6 7 5 0 3 4 8 2
ASTAR_MANHATTAN	16	26.6 ms	95	63	1 6 7 5 0 3 4 8 2
UCS	20	1734.7 ms	51072	17894	7 1 2 4 8 5 6 3 0
ASTAR_MISPLACED	20	156.9 ms	3302	1815	7 1 2 4 8 5 6 3 0
ASTAR_MANHATTAN	20	89.0 ms	502	305	7 1 2 4 8 5 6 3 0
UCS	24	4372.9 ms	121912	24188	0 7 2 4 6 1 3 5 8
ASTAR_MISPLACED	24	804.1 ms	18429	8598	0 7 2 4 6 1 3 5 8
ASTAR_MANHATTAN	24	214.8 ms	1757	967	0 7 2 4 6 1 3 5 8
UCS	31	6837.2 ms	181438	25142	8 6 7 2 5 4 3 0 1
ASTAR_MISPLACED	31	6356.2 ms	145406	24443	8 6 7 2 5 4 3 0 1
ASTAR_MANHATTAN	31	2225.3 ms	21197	8861	8 6 7 2 5 4 3 0 1

Table 1 : Performance Metrics of 8-Puzzle game [6]

In the table 1, if we take a close look at the time column, we can observe that the time taken by UCS increases rapidly as the depth increases. This is expected because UCS doesn't make use of any heuristic to guide the search, and hence, it explores more nodes. However, the A* algorithms, with both Misplaced Tile Heuristic and Manhattan Distance Heuristic, performed much better in terms of time efficiency. The Manhattan Distance Heuristic consistently took less time compared to the other two at higher depths as the heuristic is more informed which reduces the search space.

On observing the Nodes Expanded column, we can see that UCS expands more nodes, significantly more at higher depths. We can clearly see the inefficiency of UCS in comparison to the A* algorithms. However, the A* with Manhattan Distance Heuristic stands out as it expands the least number of nodes across all depths which is again due to its heuristic that provides a better estimate of the cost, thereby reducing unnecessary node expansions.

I have included four graphs which I made on *"Plotly Chart Studio"* [7] using the above data. The graphs are as follows :

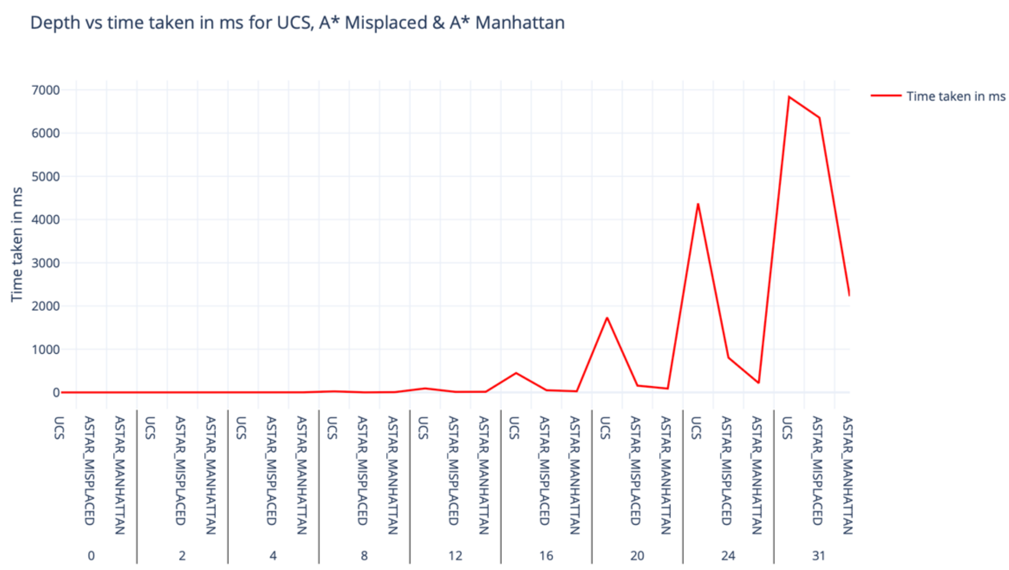

Figure 3 : Depth vs time taken in ms for UCS, A Misplaced & A* Manhattan*

If we clearly observe, in Figure 3 we can see that as the depth of the puzzle increases, the time taken by the Uniform Cost Search (UCS) algorithm to solve the puzzle raises quite sharply. This is because UCS doesn't use a heuristic to guide its search, resulting in exploring more nodes which is time consuming.

However, the time taken by A* with both the Misplaced Tile Heuristic and Manhattan Distance Heuristic doesn't increase as rapidly and the A* with Manhattan Distance Heuristic is less affected by the increase in depth, maintaining a relatively lower time increase. This is because of its heuristic as discussed above.

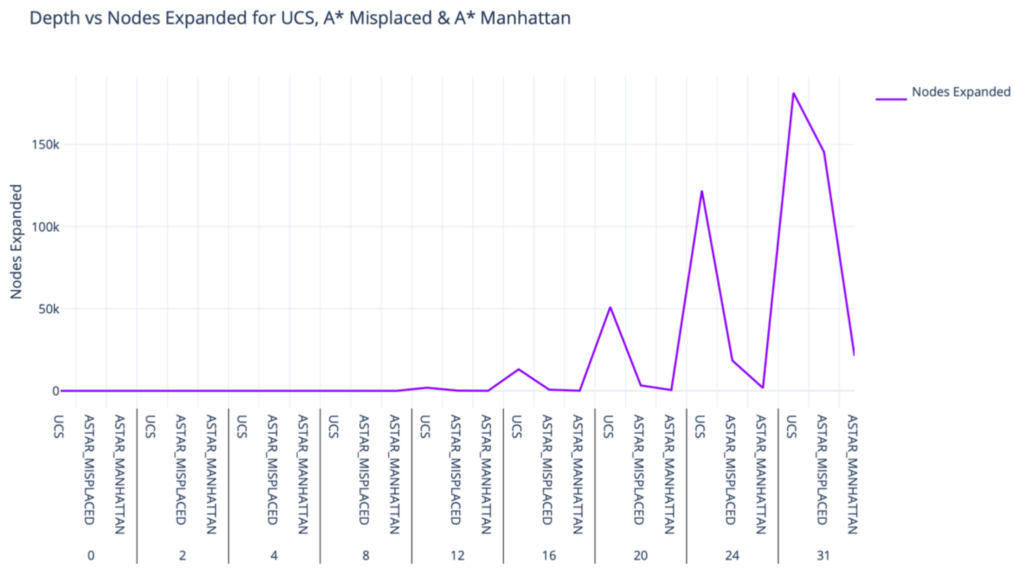

Figure 4 : Depth vs Nodes Expanded for UCS, A Misplaced & A* Manhattan*

In figure 4, we can notice some interesting patterns. As the depth of the puzzle rises, UCS expands a significantly higher number of nodes compared to the two A* algorithms. This is again due to UCS's mechanism which does not incorporate a heuristic to guide its search, leading to exploration of a larger number of nodes1.

The A* with Misplaced Tile Heuristic and the A* with Manhattan Distance Heuristic, on the other hand have a slower growth in the number of nodes expanded. The A* with Manhattan Distance Heuristic expands fewer nodes as the depth increases, which shows that the extra informed heuristic which also uses the moves to the goal state is helping the algorithm in expanding fewer nodes.

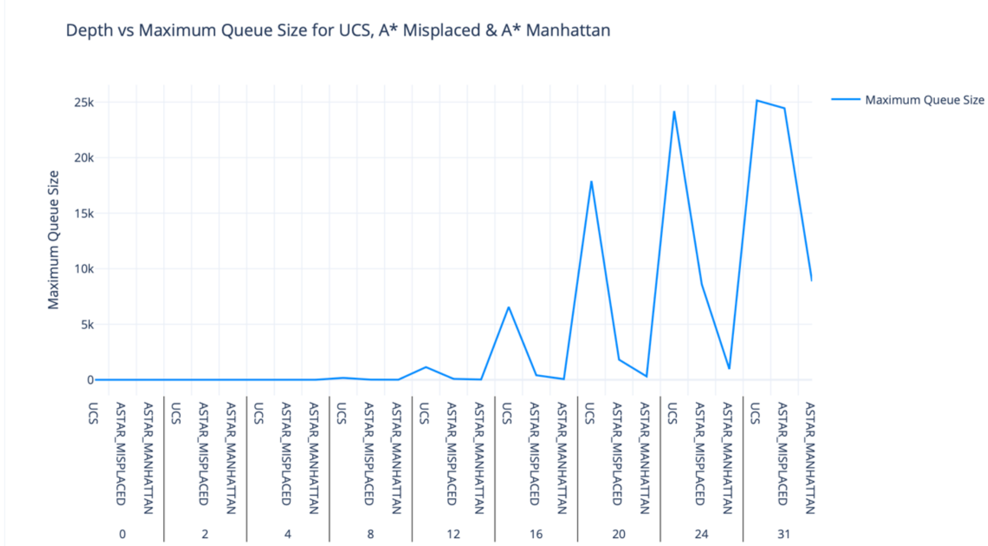

Figure 5 : Depth vs Maximum Queue Size for UCS, A Misplaced & A* Manhattan*

In the above figure 5, we can see that Uniform Cost Search (UCS) algorithm results in the highest max queue size as the depth increases which indicates its exhaustive exploration of nodes.

A* algorithms, both with Misplaced Tile and Manhattan Distance Heuristics, do relatively well in managing the queue size due to the heuristic guidance. We can also clearly see that the A* with Manhattan Distance Heuristic got keeps the queue size comparatively lower. This can be attributed to its heuristic that accurately estimates the cost, thereby reducing the number of nodes to be explored.

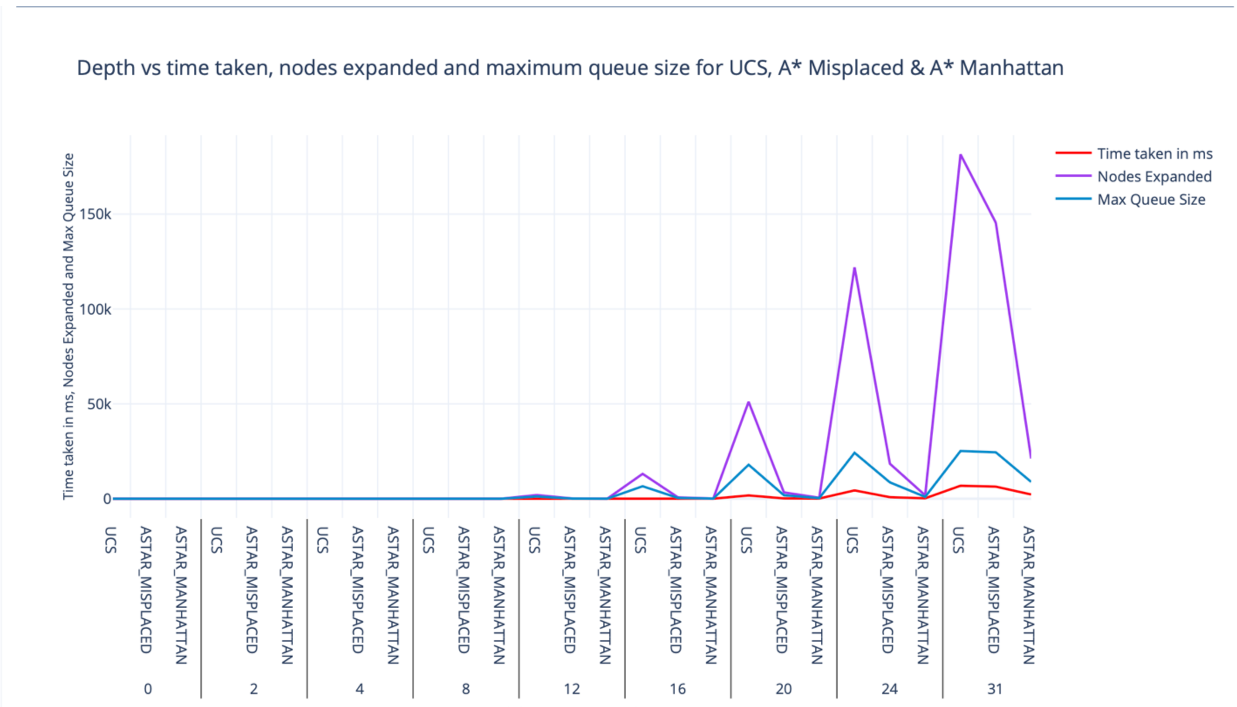

Figure 6 : Depth vs time taken, nodes expanded and maximum queue size for UCS, A Misplaced & A* Manhattan*

Looking at Figure 6, we can observe a clear pattern across the three algorithms - Uniform Cost Search (UCS), A* with Misplaced Tile Heuristic, and A* with Manhattan Distance Heuristic. As the depth of the puzzle increases, all three metrics (time taken, nodes expanded, and maximum queue size) show a similar pattern of increase for UCS, demonstrating its exhaustive nature.

However, for the A* algorithms, we notice a comparatively less sharp increase in these values. Consider A* with Manhattan Distance where its Heuristic performs notably better as it has efficient heuristic that helps guide the search to the goal state with fewer expansions.

From the graph above we can effectively see that an informed search strategy, like A*, can outperform an uninformed one like UCS in terms of time efficiency, search space and memory utilization, especially as the complexity of the problem (depth) increases.

Conclusion

It's clear from the project and the data that when it comes to solving the 8-puzzle game, informed search strategies tend to outperform uninformed ones. As we examined Uniform Cost Search (UCS), A* with Misplaced Tile heuristic, and A* with Manhattan Distance heuristic, it was evident that both A* algorithms, equipped with their heuristics, were more efficient in terms of time, nodes expanded, and maximum queue size as compared to UCS. The A* with Manhattan Distance heuristic was found to be consistently superior due to its more informed heuristic.

I've also observed that as the puzzle depth increased, the time taken and the number of nodes expanded by the UCS algorithm increased significantly as it doesn't make use of any heuristic to guide its search. On the other hand, the A* algorithms managed to keep their time and nodes expanded to a minimum, even at higher depths. The Manhattan Distance heuristic was highly effective due to the fact that it takes into account not just the misplaced tiles but also the number of moves needed to reach the goal state. This additional information helps to guide the search more accurately, thus reducing the search space and the time taken to reach the solution.

Traceback of an Easy Puzzle (Depth = 4)

```
anirudhtulasi@Anirudhs-Air:~/Desktop/AI/Final
└ clear
 ~/Desktop/AI/Final                                              09:41:25 PM
└ python3 search.py
Welcome to my 8-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create your own.
2
Enter your puzzle, using a zero to represent the blank. Please only enter valid 8-puzzles. Enter the puzzle demilimiting the numbers with a space. Type RETURN only when finished.
Example for Input : Three Numbers per line as shown below
8 6 7
2 5 4
3 0 1
Enter the 1 row: 1 2 3
Enter the 2 row: 5 0 6
Enter the 3 row: 4 7 8
Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile Heuristic, or (3) the Manhattan Distance Heuristic.
3
A* with Manhattan Distance heuristic Solution:
Step 0:
[[1 2 3]
 [5 0 6]
 [4 7 8]]
Step 1:
[[1 2 3]
 [0 5 6]
 [4 7 8]]
Step 2:
[[1 2 3]
 [4 5 6]
 [0 7 8]]
Step 3:
[[1 2 3]
 [4 5 6]
 [7 0 8]]
Step 4:
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Depth of solution: 4
Path cost: 4
Number of nodes expanded: 4
Max queue size: 6
Time taken: 6.511211395263672 ms
```

Traceback of an hard Puzzle (Depth = 16)

```
anirudhtulasi@Anirudhs-Air:~/Desktop/AI/Final
~/Desktop/AI/Final                                              33s  09:42:02 PM
python3 search.py
Welcome to my 8-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create your own.
2
Enter your puzzle, using a zero to represent the blank. Please only enter valid 8-puzzles. Enter the puzzle demilimiting the numbers with a space. Type RETURN only when finished.
Example for Input : Three Numbers per line as shown below
8 6 7
2 5 4
3 0 1
Enter the 1 row: 1 6 7
Enter the 2 row: 5 0 3
Enter the 3 row: 4 8 2
Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile Heuristic, or (3) the Manhattan Distance Heuristic.
3
A* with Manhattan Distance heuristic Solution:
Step 0:
[[1 6 7]
 [5 0 3]
 [4 8 2]]
Step 1:
[[1 6 7]
 [5 3 0]
 [4 8 2]]
Step 2:
[[1 6 0]
 [5 3 7]
 [4 8 2]]
Step 3:
[[1 0 6]
 [5 3 7]
 [4 8 2]]
```

//Removed steps 4 to 11 in screenshots to save some space.

```
Step 12:
[[1 2 3]
 [5 0 6]
 [4 7 8]]
Step 13:
[[1 2 3]
 [0 5 6]
 [4 7 8]]
Step 14:
[[1 2 3]
 [4 5 6]
 [0 7 8]]
Step 15:
[[1 2 3]
 [4 5 6]
 [7 0 8]]
Step 16:
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Depth of solution: 16
Path cost: 16
Number of nodes expanded: 95
Max queue size: 63
Time taken: 29.451847076416016 ms

~/Desktop/AI/Final                                              35s  09:45:24 PM
```

Code:

node.py | eight_puzzle.py | search.py | test_cases.csv

node.py > ...

```python
import numpy as np

class Node:
    def __init__(self, state, parent, action, path_cost):
        self.state = state
        self.parent = parent
        self.action = action
        if parent is not None:
            self.path_cost = parent.path_cost + path_cost
            self.depth = parent.depth + 1  # Increment depth from parent node
        else:
            self.path_cost = path_cost
            self.depth = 0  # Root node depth is 0

    def __eq__(self, other):
        return isinstance(other, Node) and np.array_equal(self.state, other.state)

    def __lt__(self, other):
        return self.path_cost < other.path_cost

```

node.py | eight_puzzle.py | search.py | test_cases.csv

eight_puzzle.py > ...

```python
import numpy as np
from copy import deepcopy
from node import Node

class EightPuzzle:
    def __init__(self, initial_state, goal_state):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.actions_list = ["UP", "DOWN", "LEFT", "RIGHT"]

    def actions(self, state):
        possible_actions = []
        zero_index = np.where(np.array(state) == 0)

        if zero_index[1] > 0:  # LEFT action possible
            possible_actions.append("LEFT")
        if zero_index[1] < 2:  # RIGHT action possible
            possible_actions.append("RIGHT")
        if zero_index[0] > 0:  # UP action possible
            possible_actions.append("UP")
        if zero_index[0] < 2:  # DOWN action possible
            possible_actions.append("DOWN")

        return possible_actions

    def result(self, state, action):
        new_state = deepcopy(state)
        zero_index = np.where(np.array(state) == 0)

        if action == "UP":
            new_state[zero_index[0][0]][zero_index[1][0]], new_state[zero_index[0][0] - 1][zero_index[1][0]] = new_state[zero_index[0][0] - 1][zer
        elif action == "DOWN":
            new_state[zero_index[0][0]][zero_index[1][0]], new_state[zero_index[0][0] + 1][zero_index[1][0]] = new_state[zero_index[0][0] + 1][zer
        elif action == "LEFT":
            new_state[zero_index[0][0]][zero_index[1][0]], new_state[zero_index[0][0]][zero_index[1][0] - 1] = new_state[zero_index[0][0]][zero_ir
        elif action == "RIGHT":
            new_state[zero_index[0][0]][zero_index[1][0]], new_state[zero_index[0][0]][zero_index[1][0] + 1] = new_state[zero_index[0][0]][zero_ir

        return new_state

    def goal_test(self, state):
        return np.array_equal(state, self.goal_state)

    def step_cost(self, parent_state, action):
```

node.py | eight_puzzle.py | search.py | test_cases.csv

eight_puzzle.py > ...

```python
    def step_cost(self, parent_state, action):
    # Assuming each action has a cost of 1
        return 1

    def child_node(self, parent, action):
        child_state = self.result(parent.state, action)
        return Node(child_state, parent, action, self.step_cost(parent.state, action))


    def h(self, state, heuristic):
        distance = 0
        if heuristic == "misplaced_tiles":
            return np.sum(state != self.goal_state) - 1  # subtract 1 because we don't count the blank space
        elif heuristic == "manhattan_distance":
            distance = 0
            for i in range(1, 9):  # 1-8 for 8-puzzle
                actual_position = np.array(np.where(state == i))
                goal_position = np.array(np.where(self.goal_state == i))
                distance += np.sum(np.abs(actual_position - goal_position))
        return distance

    def solution(self, node):
        path = []
        while node is not None:
            path.append(node)
            node = node.parent
        return path[::-1]  # reverse
```

node.py | eight_puzzle.py | search.py | test_cases.csv

search.py > ...

```python
import heapq
import numpy as np
from eight_puzzle import EightPuzzle
from node import Node
import time

def general_search(problem, heuristic=None):
    node = Node(problem.initial_state, None, None, 0)
    if problem.goal_test(node.state):
        return problem.solution(node), 0, 1  # Start the max queue size from 1

    frontier = []
    frontier_set = set()  # A set to keep track of states in the frontier
    heapq.heappush(frontier, (node.path_cost + (problem.h(node.state, heuristic) if heuristic else 0), node))
    frontier_set.add(tuple(map(tuple, node.state.tolist())))  # Add the node's state to frontier_set
    explored = set()
    nodes_expanded = 0
    max_queue_size = 1  # Initialize the max queue size

    while frontier:
        _, node = heapq.heappop(frontier)
        frontier_set.remove(tuple(map(tuple, node.state.tolist())))  # Remove the node's state from frontier_set
        if problem.goal_test(node.state):
            return problem.solution(node), nodes_expanded, max_queue_size

        explored.add(tuple(map(tuple, node.state.tolist())))  # convert np.array to tuple for hashing
        nodes_expanded += 1

        for action in problem.actions(node.state):
            child = problem.child_node(node, action)
            child_tuple = tuple(map(tuple, child.state.tolist()))

            if child_tuple not in explored and child_tuple not in frontier_set:
                heapq.heappush(frontier, (child.path_cost + (problem.h(child.state, heuristic) if heuristic else 0), child))
                frontier_set.add(child_tuple)  # Add the child's state to frontier_set

                # If the current size of the queue is larger than the max queue size, update the max queue size
                if len(frontier) > max_queue_size:
                    max_queue_size = len(frontier)

    return None, nodes_expanded, max_queue_size
```

node.py | eight_puzzle.py | search.py | test_cases.csv

search.py > ...

```python
def print_solution(solution):
    for i, node in enumerate(solution):
        print(f"Step {i}:")
        print(node.state)

def get_state_from_user():
    state = []
    for i in range(1, 4):
        row = list(map(int, input(f"Enter the {i} row: ").split()))
        state.append(row)
    return np.array(state)

def main():
    print("Welcome to my 8-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create your own.")
    choice = input()

    if choice == '1':
        initial_state = np.array([[8, 6, 7], [2, 5, 4], [3, 0, 1]])
        goal_state = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
    elif choice == '2':
        print("Enter your puzzle, using a zero to represent the blank. Please only enter valid 8-puzzles. Enter the puzzle demilimiting the number
        print("Example for Input : Three Numbers per line as shown below")
        print("8 6 7")
        print("2 5 4")
        print("3 0 1")
        initial_state = get_state_from_user()
        goal_state = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
    else:
        print("Invalid choice.")
        return

    problem = EightPuzzle(initial_state, goal_state)

    print("Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile Heuristic, or (3) the Manhattan Distance Heuristic.")
    algo_choice = input()

    if algo_choice == '1':
        print("Uniform Cost Search Solution:")
        start_time = time.time()
        solution, expanded_nodes, max_queue_size = general_search(problem)
    elif algo_choice == '2':
        print("A* with Misplaced Tile heuristic Solution:")
        start_time = time.time()
        solution, expanded_nodes, max_queue_size = general_search(problem, "misplaced_tiles")
```

```
        start_time = time.time()
        solution, expanded_nodes, max_queue_size = general_search(problem, "misplaced_tiles")
    elif algo_choice == '3':
        print("A* with Manhattan Distance heuristic Solution:")
        start_time = time.time()
        solution, expanded_nodes, max_queue_size = general_search(problem, "manhattan_distance")
    else:
        print("Invalid choice.")
        return

    end_time = time.time()

    if solution is not None:
        print_solution(solution)
        print(f"Depth of solution: {solution[-1].depth}")
        print(f"Path cost: {solution[-1].path_cost}")
        print(f"Number of nodes expanded: {expanded_nodes}")
        print(f"Max queue size: {max_queue_size}")
    else:
        print("No solution found.")

    print(f"Time taken: {(end_time - start_time) * 1000} ms")  # Multiply by 1000 to get time in milliseconds

if __name__ == "__main__":
    main()
```

- Code uploaded on : https://github.com/anirudhtulasi/CS205P1

References

[1] Daniel Ratner, Manfred Warmuth, The (n2−1)-puzzle and related relocation problems, Journal of Symbolic Computation, Volume 10, Issue 2, 1990, Pages 111-137, ISSN 0747-7171

[2] Deniz Gürkaynak, 8-Puzzle Solver (https://deniz.co/8-puzzle-solver/)

[3] Dr.Keogh, Project Handout (https://rb.gy/kios5)

[4] Dr.Keogh, Project Handout (https://rb.gy/kios5)

[5] Dr.Keogh, Project Handout (https://rb.gy/kios5)

[6] Deniz Gürkaynak, 8-Puzzle Solver (https://deniz.co/8-puzzle-solver/)

[7] Plotly Chart Studio (https://chart-studio.plotly.com/)