CS205 Project 1 : 8 Puzzle

Guanqiu Wu
SID: 862395013
Email: gwu034@ucr.edu
Date: May 15 2023

Basic Information:

(1) For this project I used python3 to complete.
(2) The URL of my code is https://github.com/RSGooner/CS205-8PUZZLE
(3) In order to complete this project, I have referred to many professors' slides and some online materials, which will be presented in the reference.

Outline of The Report:

Introduction

8 puzzle problems is a slider puzzle based on a 3x3 grid with 9 squares, 8 of which are marked with numbers 1 to 8 and one empty space. The player needs to make the 8 squares present a target order by moving the squares. The general default target order is shown in the figure 1. Each step the player can choose one square and space to swap, only up and down and left and right not diagonal swap. There are also other sizes and types of slider puzzles that are similar to them, such as 15 Puzzle problems, 35 Puzzle problems, and Klotski. They provide a fun and challenging activity, and to solve such problems requires strong logical thinking and problem solving skills, so they are widely found in mechanical or electronic device puzzle games.

Figure 1. 8 Puzzle

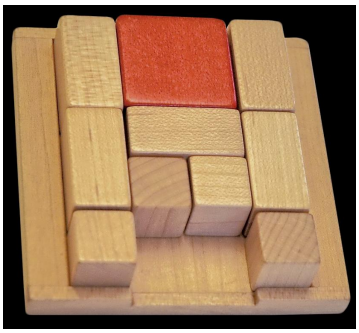

Figure 2. Klotski Puzzle

The 8 puzzle problem aims to find a relatively shortest path to the target state, and different algorithms can be used to solve the problem using a program. For this project, three different A* algorithms with different heuristic functions were chosen to solve the problem: UCS, The Misplaced Tile Heuristic, and The Manhattan Distance Heuristic, and the performance of these three heuristic functions to solve the same initial state was compared. I used python3 to implement this project, and the full code will be available at the end of this report.

A* Algorithm

The A* algorithm is a direct search method for solving the shortest path in a static road network most efficiently, and is also a common heuristic algorithm for many other problems.

The formula is expressed as: f(n) = g(n) + h(n).where f(n) is the minimum cost estimate from the initial state through state n to the target state, and
g(n) is the minimum cost from the initial state to state n in the state space, and
h(n) is the minimum cost estimate of the path from state n to the target state.

The key to ensuring that the shortest path condition is found is the selection of the estimation function h(n).

Comparison of the three heuristic functions

Uniform Cost Search (UCS)

UCS is based on the extension of BFS, and the main difference between them is that UCS maintains a priority queue through which the path to the target node is guaranteed to be the shortest. When h(n) is always 0, the A* algorithm will degenerates to the UCS algorithm.

The Misplaced Tile Heuristic

The misplaced tile heuristic is a common heuristic for solving sliding tile puzzles. A heuristic is an estimation or rule of thumb that guides the search for a solution by providing an approximate measure of how close a given state is to the target state.

In the case of the misplaced tile heuristic, the idea is to count the number of tiles that are not in the correct state with respect to the target state. Each tile that is not in the correct state is considered to be a misplaced tile. It is assumed that the fewer misplaced tiles there are, the closer the current state is to the goal state.

The Manhattan Distance Heuristic

In mathematics, there are many formulas to calculate distances, such as Euclidean distance, Marxian distance, etc. In this project I used the Manhattan distance. For a town street with a regular layout of due south and north, east and west directions, the distance from one point to another is exactly the distance traveled in the north-south direction plus the distance traveled in the east-west direction, hence the Manhattan distance is also called the cab distance.

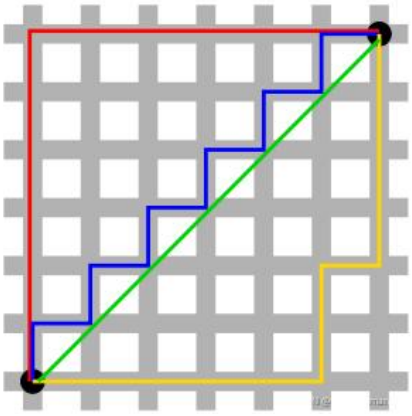

Figure 3. the blue line is Euclidean distance,
the other three lines are Manhattan distance.

Floating point operations are expensive, slow and error-prone. If you use Euclidean distances directly from AB), you have to do floating point operations, if you use AC and CB, you just have to calculate addition and subtraction, which greatly increases the speed of operations, and there is no error no matter how many times you accumulate the operations.

Performance comparison of three heuristic functions

I have pre-set 8 different difficulty initial states that you can choose from when running the code. They are:

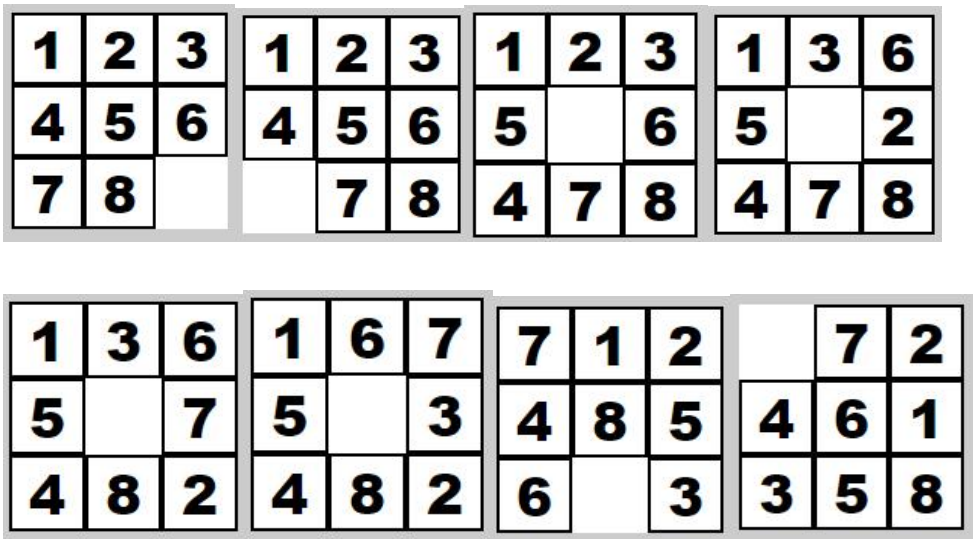

Figure 4. 8 different difficulties of the initial state of the 8 puzzle problems

Here I have chosen the initial state with difficulty level 2 and 8 for the performance test of three different heuristic functions, and the test metrics are number of nodes expanded and max queue size.

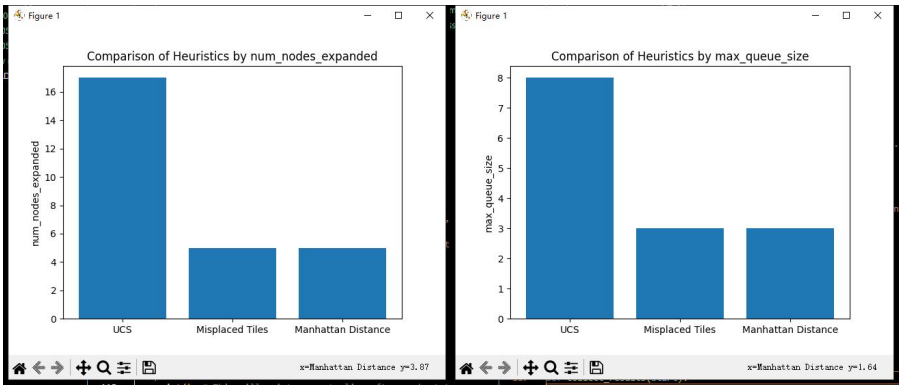

Figure 5. number of nodes expanded and max queue size of level 2 puzzle.

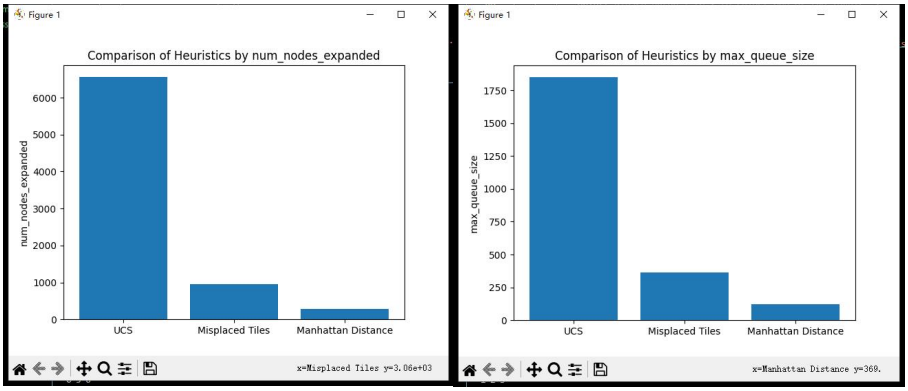

Figure 6. number of nodes expanded and max queue size of level 2 puzzle.

Using the matplotlib library in python to plot histograms and compare the performance of the three methods to solve the default problem, it is clear that the different heuristic functions have a huge impact on the performance. In the absence

of heuristic functions, both problems have significantly more extended nodes and maximum queue sizes, with two values of 17 and 8 for the difficulty 2 problem and two values greater than 6000 and 1750 for the difficulty 8 problem, respectively, however, there is a significant drop after using the misplaced tile heuristic and a further drop after using the Euclidean distance heuristic.

Conclusion

In solving the eight-digit problem, the performance of the heuristic search algorithm depends mainly on the chosen heuristic function. We consider three different heuristic functions: uniform cost search, misplaced tiles, and Manhattan distance.

(1)Uniform cost search: this is actually the case of no heuristic function, which is the ordinary breadth-first search. For simple problems, this approach may find a solution, but for more complex problems (e.g., 15-puzzle with random initial states), it may require extremely long time and huge memory.

(2)Misplaced tiles: This heuristic function simply counts the number of misplaced tiles in the current state. It is better than uniform cost search because it introduces a preference in the search: those states with fewer misplaced tiles will be given priority. However, this heuristic function is not always good, because it does not take into account the actual number of steps needed to reach the target state.

(3)Manhattan distance: This is the most commonly used heuristic function in the octet problem and it calculates the minimum number of steps required for each tile to get from its current position to the target position (moving up and down, left and right in the grid). This heuristic function is better than the misaligned tiles because it gives a more accurate expected cost of reaching the target state.

Overall, while the uniform cost search and the misplaced tile heuristic can work in some cases, the Manhattan distance heuristic performs well in a wider range of cases, especially for complex puzzle problems. However, no one heuristic function is optimal in all cases, and the choice of which heuristic function to use depends on the specific problem and scenario.

Reference

[1] AI classical search algorithm knowledge points
https://blog.csdn.net/qq_39753778/article/details/104779143

[2] Python solves the eight-digit problem using BFS DFS UCS Greedy A*
algorithmhttps://blog.csdn.net/weixin_53525090/article/details/127290979

[3] Python various drawing
https://blog.csdn.net/Gou_Hailong/article/details/120089602

The following is a traceback of the 2 level difficulty puzzle

If you want to use default puzzle, please enter 1, if you want to set the puzzle by yourself, please input 2 :1

The preset puzzle has 8 different levels of difficulty, enter 1 to 8 to select one of the

levels:2

Please select the heuristic function (input 1 for UCS, 2 for misplaced, 3 for manhattan):1

Puzzle has solved!

1 2 3

4 5 6

0 7 8

1 2 3

4 5 6

7 0 8

1 2 3

4 5 6

7 8 0

Solution depth: 2

Number of nodes expanded: 17

Max queue size: 8

The following is a traceback of the 8 level difficulty puzzle

If you want to use default puzzle, please enter 1, if you want to set the puzzle by yourself, please input 2 :2

Please enter the initial state of the puzzle, 0 for empty space, and enter 9 numbers separated by spaces.

0 7 2 4 6 1 3 5 8

Please select the heuristic function (input 1 for UCS, 2 for misplaced, 3 for manhattan):3

Puzzle has solved!

0 7 2

4 6 1

3 5 8

7 0 2

4 6 1

3 5 8

7 2 0

4 6 1

3 5 8

::: //I deleted 50 steps to save space.

1 2 3

4 5 6

7 0 8

1 2 3

4 5 6

7 8 0

Solution depth: 58

Number of nodes expanded: 283

Max queue size: 120

```
from heapq import heappop, heappush
import matplotlib.pyplot as plt


# Four Directions Move
MOVES = [(0, -1), (0, 1), (-1, 0), (1, 0)]

def manhattan_distance(state, goal):
    """Return the sum of the Manhattan distances of the tiles from their goal positions"""
    size = 3
    distances = []
    for n in range(1, size*size):
        i = state.index(n)
        g = goal.index(n)
        distance = abs(i // size - g // size) + abs(i % size - g % size)
        distances.append(distance)
    return sum(distances)



def number_of_misplaced_tiles(state, goal):
    """Return the number of misplaced tiles"""
    return sum(s != g for (s, g) in zip(state, goal))

def find_neighbors(state):
    """Return the states reachable from state"""
    size = 3
    i = state.index(0)    # The position of the blank
    (x, y) = divmod(i, size)
    for (dx, dy) in MOVES:
        (nx, ny) = (x + dx, y + dy)
        if 0 <= nx < size and 0 <= ny < size:
            next_state = state.copy()
            next_state[i], next_state[nx * size + ny] = next_state[nx * size + ny], next_state[i]
            yield (1, next_state)

def reconstruct_path(pre_state, state):
    """Return the path from the start state to state"""
    return [] if state is None else reconstruct_path(pre_state, pre_state[tuple(state)]) + [state]

def solve_puzzle(start, heuristic):
    """Main function to solve the puzzle."""
    # Define the goal state
    goal = list(range(1, 9)) + [0]
```

```
    # Define the heuristic function. I use lambda to avoid TyperError
    if heuristic == 1:
        h = lambda s: 0
    elif heuristic == 2:
        h = lambda s: number_of_misplaced_tiles(s, goal)
    elif heuristic == 3:
        h = lambda s: manhattan_distance(s, goal)

    # Priority queue, where the priority (score) is the first element
    priority_value = h(start)    # calculateh(n)
    queue = [(priority_value, start)]    # edit the queue



    # Dictionary of {state: predecessor}
    pre_state = {tuple(start): None}

    max_queue_size = 1    # Keep track of max queue size

    while queue:
        (priority, state) = heappop(queue)
        if state == goal:
            path = reconstruct_path(pre_state, state)
            solution_depth = len(path) - 1
            num_nodes_expanded = len(pre_state)
            solution_info = {
                'solution_depth': solution_depth,
                'num_nodes_expanded': num_nodes_expanded,
                'max_queue_size': max_queue_size,
            }
            return path, solution_info
        for (cost, next_state) in find_neighbors(state):
            if tuple(next_state) not in pre_state:
                priority = cost + h(next_state)
                heappush(queue, (priority, next_state))
                pre_state[tuple(next_state)] = state
                max_queue_size = max(max_queue_size, len(queue))    # Update max queue size if necessary
    return []



def input_puzzle():
    """Prompt the user to input the initial state of the puzzle and the heuristic to use"""
    puzzle_type = int(input("If you want to use default puzzle, please enter 1, if you want to set the puzzle by yourself, please input 2 :"))
```

```
    if puzzle_type == 1:
        difficulty_type = int(input("The preset puzzle has 8 different levels of difficulty, enter 1 to 8 to select one of the levels:"))
        if difficulty_type == 1:
            start = [1, 2, 3, 4, 5, 6, 7, 8, 0]
        elif difficulty_type == 2:
            start = [1, 2, 3, 4, 5, 6, 0, 7, 8]
        elif difficulty_type == 3:
            start = [1, 2, 3, 5, 0, 6, 4, 7, 8]
        elif difficulty_type == 4:
            start = [1, 3, 6, 5, 0, 2, 4, 7, 8]
        elif difficulty_type == 5:
            start = [1, 3, 6, 5, 0, 7, 4, 8, 2]
        elif difficulty_type == 6:
            start = [1, 6, 7, 5, 0, 3, 4, 8, 2]
        elif difficulty_type == 7:
            start = [7, 1, 2, 4, 8, 5, 6, 3, 0]
        elif difficulty_type == 8:
            start = [0, 7, 2, 4, 6, 1, 3, 5, 8]
        else:
            print("Please enter 1 to 8 to select one of the levels.")
            return input_puzzle()
    else:
        print("Please enter the initial state of the puzzle, 0 for empty space, and enter 9 numbers separated by spaces.")
        start = list(map(int, input().split()))
    heuristic = int(input("Please select the heuristic function (input 1 for UCS, 2 for misplaced, 3 for manhattan):"))
    return start, heuristic


def print_state(state):
    """Print the state of the puzzle"""
    size = 3
    for i in range(size):
        for j in range(size):
            print(state[i*size + j], end=" ")
        print()
    print()    # This will print an empty line after each state

def collect_results(start):
    """Collect results for all heuristics and plot them."""
    results = {}
    metrics = ['solution_depth', 'num_nodes_expanded', 'max_queue_size']
```

```
    for heuristic in [1, 2, 3]:    # Iterate over all heuristics
        _, results[heuristic] = solve_puzzle(start, heuristic)

    # Prepare data for plotting
    data_to_plot = {metric: [] for metric in metrics}
    for heuristic, result in results.items():
        for metric in metrics:
            data_to_plot[metric].append(result[metric])

    # Plot the results
    for metric, data in data_to_plot.items():
        plt.figure()
        plt.bar(['UCS', 'Misplaced Tiles', 'Manhattan Distance'], data)
        plt.title('Comparison of Heuristics by {}'.format(metric))
        plt.ylabel(metric)
        plt.show()


def main():
    start, heuristic = input_puzzle()
    path, solution_info = solve_puzzle(start, heuristic)
    if path:
        print("Puzzle has solved!")
        for state in path:
            print_state(state)

        print(f"Solution depth: {solution_info['solution_depth']}")
        print(f"Number of nodes expanded: {solution_info['num_nodes_expanded']}")
        print(f"Max queue size: {solution_info['max_queue_size']}")
    else:
        print("Fail to solve.")

    collect_results(start)


main()
```