STA303 Assignment3 Report

12110208 宋子烨

Part 1: Description

To solve this problem, I consider their are two ways to find the shortest path: Bread First Search and Deep First Search based on A* algorithm.

class Node

I also construct a class called Node to simplify this problem. the code is as below:
class Node:

```
class Node:
    use_default_comparison = True
    def __init__(self, state, parent=None, cost=0, heuristic=0):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic
    def __lt__(self, other):
        if Node.use_default_comparison:
            return (self.cost + 2 * self.heuristic) < (other.cost + 2 *
other.heuristic)
        else:
            return (self.cost + 0.1 * self.heuristic) < (other.cost + 0.1 *
other.heuristic)
```

In this class, we define 4 member variables:

- **state**: represent the class station, to record current station when running.
- **parent**: represent the parent station, to record where the current station is from.
- **cost**: represent the total distance between the start station and current station, as the g(x) in A* algorithm.(Caution: the cost is not the straight-line distance between the start station and current station, it records all the path distances)
- **heuristic**: represent the distance between the current station and the end station.

heuristic functions

As for heuristic functions, I select two of them. one is manhattan_distance and the other is standard distance. The code is as follows:

```
def standard_distance(station1, station2):
    return (station1.position[0] - station2.position[0]) ** 2 +
(station1.position[1] - station2.position[1]) ** 2

def manhattan_distance(station1, station2):
    distance1 = abs(station1.position[0] - station2.position[0])
    distance2 = abs(station1.position[1] - station2.position[1])
    return distance1 + distance2
```

The effects between the two functions will discussed later.

BFS

I prefer to illustrate the code first:

```
def BFS(start_station, end_station, my_map):
    visited = set()
    start_node = Node(state=start_station, cost=0,
heuristic=heuristic(start_station, end_station))
    queue = deque([start_node])  # construct deque to store and get Node
    while True:
        if not queue:
            break
        current_node = queue.popleft()
        visited.add(current_node.state.name)
        if current_node.state.id == end_station.id:
            return_path = []
            while True:
                return_path.append(current_node.state.name)
                current_node = current_node.parent
                if current_node is None:
                    break
            changedList = return_path[::-1]
            return changedList
        else:
            visited.add(current_node.state.name)
            for neighbor in get_neighbors(current_node.state):
                if neighbor not in visited:
                    if heuristic(my_map[neighbor], end_station) <
current_node.heuristic:
                        neighbor_node=Node(state=my_map[neighbor],
parent=current_node,cost=current_node.cost+heuristic(current_node.state,my_map[ne
ighbor]),heuristic=heuristic(my_map[neighbor], end_station))
                            queue.append(neighbor_node)

    return []  # no path found
```

First we create a set to record all the station we have visited and create a node using start_station. Moreover, a queue is requisite to store or get node. Then we create a circulation to get all the neighbors of the current_station. If one of the neighbors has less distance with the end_station than the current_station, I make the queue to store it. If the current_station is the end_station, we return the list with the variable Node.parent.

As you can see, the deficiency of BFS is when it meet the end_station, it just returns. As a result, if there is a better path with more stations but less distance, the BFS algorithm can't get it. I will discuss it later with a practical way.

A* algorithm

```
frontier = PriorityQueue()
start_node = Node(state=start_station, cost=0, heuristic=heuristic(start_station,
end_station))
frontier.put(start_node)
explored = set()
while True:
    current_node = frontier.get()
    if current_node.state.id == end_station.id:
        return_path = []
        while True:
            return_path.append(current_node.state.name)
            current_node = current_node.parent
            if current_node is None:
                break
        return return_path
    else:
        explored.add(current_node.state.name)
        for neighbor in get_neighbors(current_node.state):  # 获取当前节点的相邻节点
            if neighbor not in explored:
                neighbor_node = Node(state=my_map[neighbor], parent=current_node,
                                     cost=current_node.cost +
heuristic(current_node.state, my_map[neighbor]),  #
                                     heuristic=heuristic(my_map[neighbor],
end_station))
                frontier.put(neighbor_node)
```

The only difference between A* algorithm and BFS is that A* algorithm use the PriorityQueue instead. PriorityQueue always give the best station as the next. The strategy is the def _lt_ in the class node.
The f(x) is:

$$self.cost + 2 * self.heuristic$$

In this function, we show more preference on the heuristic, which is the distance between the current_station and end_station.

AdjustBFS and AdjustA* algorithm

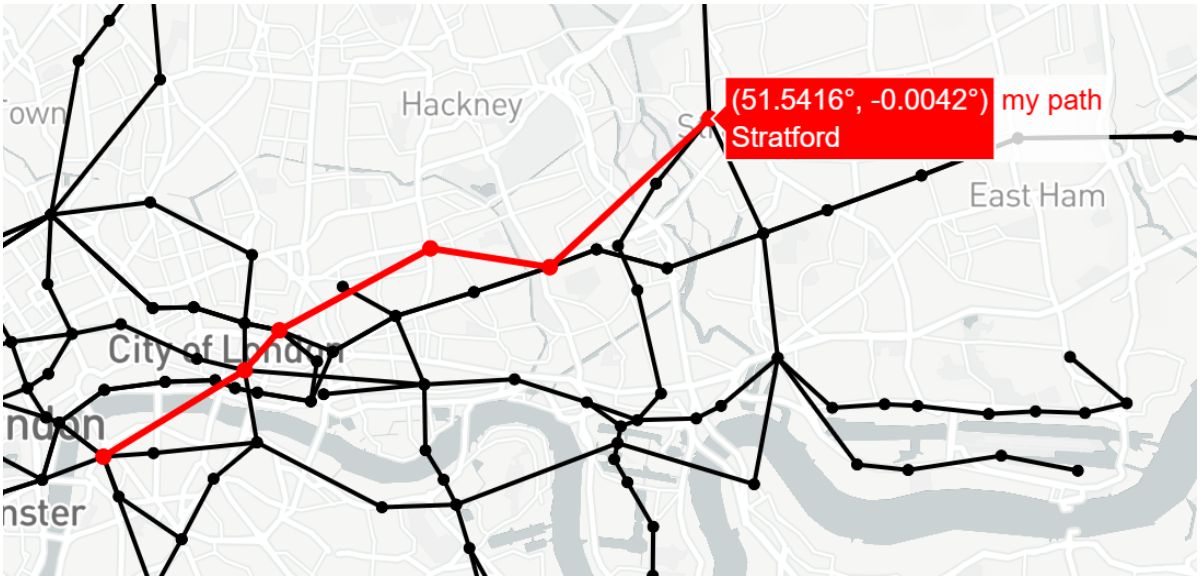
I enhance and promote these two algorithms with enlarging the scale of the search. For example, if we find a path with BFS, it will not return immediately. the circulation will go on until we find n paths, where n is a hyperparmeter. then we choose the least distance among the n paths and return it. The code is omitted, you can check in the python file `find_shortest_path.py`.

Part 2: Performance evaluation

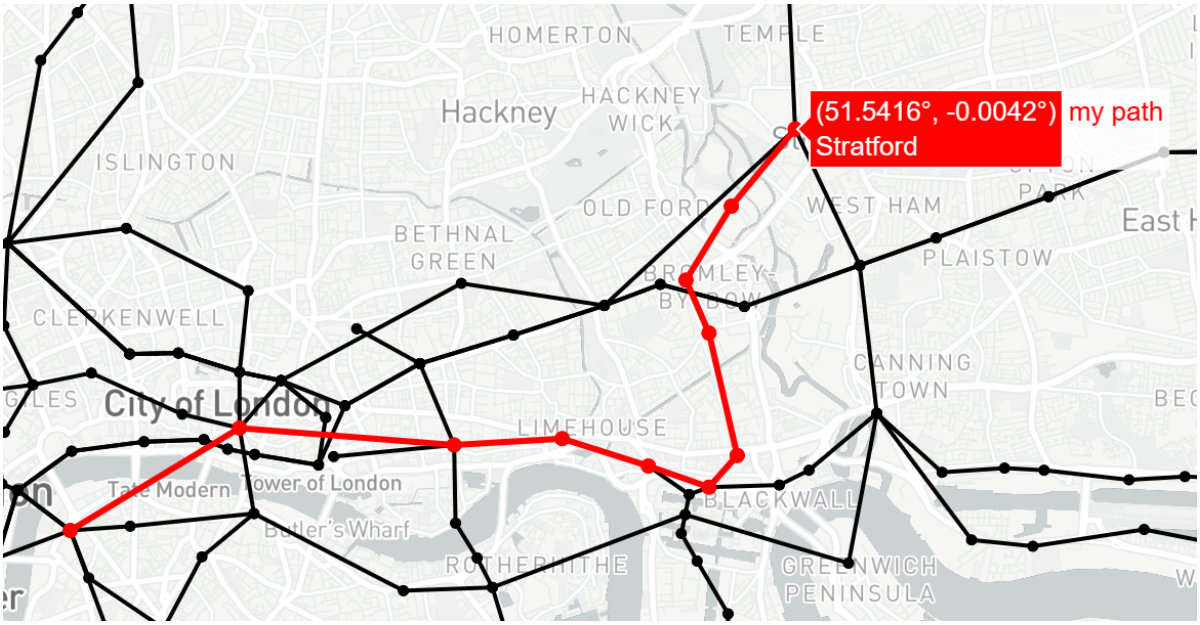
Example 1

```
python .\find_shortest_path.py "Waterloo" "Stratford"
```

Let's take the path between **Waterloo** and **Stratford** as an example. In the algorithm **BFS**, the path is demonstrated as below:



However, when we choose **A*** algorithm, the path is **different**:

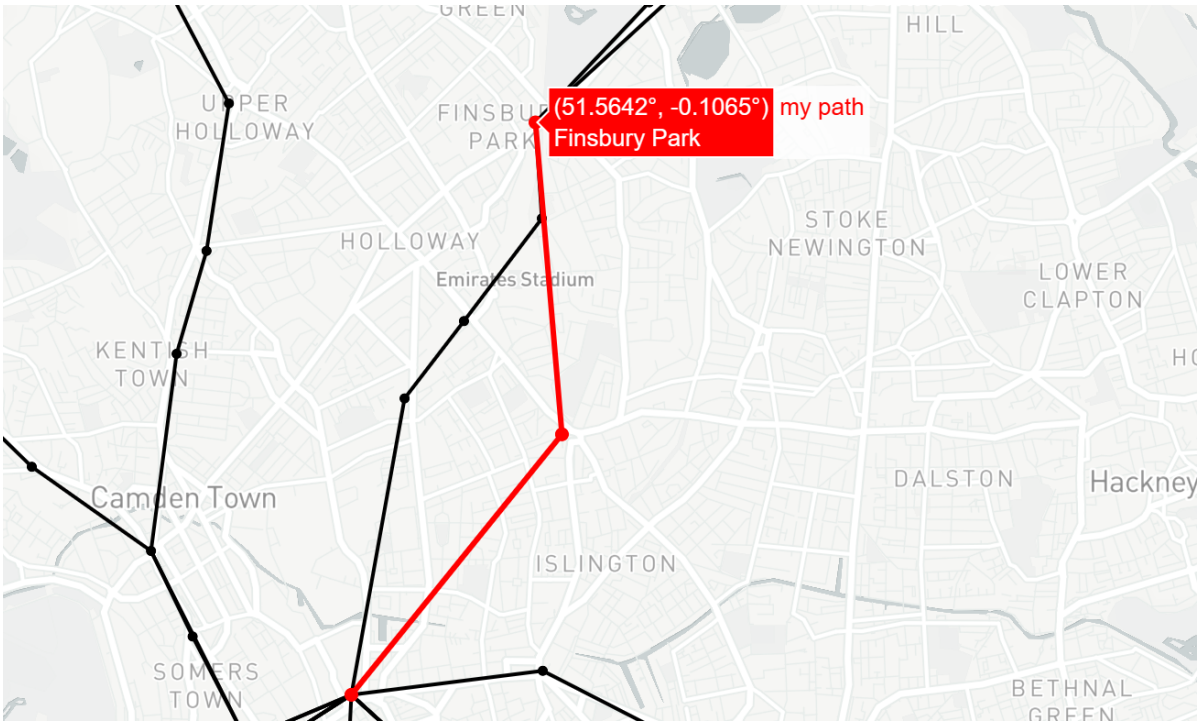


To analyze what makes the difference between the two algorithms, I **conclude** that when we meet the station called **Bank**, the second station in the two pictures, there is a divergence. **A*** algorithm choose the station **Shadwell** which is really the best under the current circumstance. However, the next several stations seem like the path isn't the best, while the **A*** algorithm don't realize it. Every step it proposes makes the heuristic smaller indeed, so it may be confident to move further. Actually, the best third station is **Liverpool Street**, which is chose by the **BFS** algorithm.

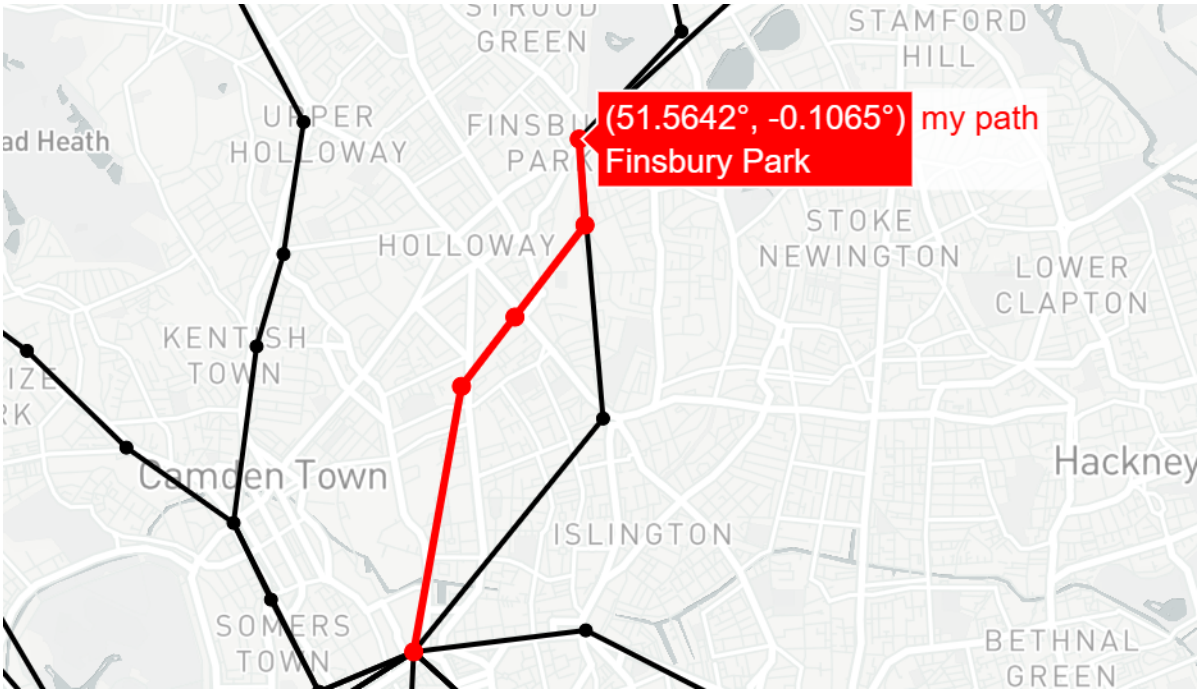
Example 2

```
python .\find_shortest_path.py "King's Cross St. Pancras" "Finsbury Park"
```

Let's take this path between **King's Cross St. Pancras** and **Finsbury Park** as an example. In the **BFS** algorithm, the path is illustrated as below:



However, when we choose **A*** algorithm, the path is **different**:



Although it is different to judge which path perform better with our eyes, the computer show the length of BFS is 0.000960570000000011,while the length of A* is 0.0004969099999999743, meaning that in this case, the A* algorithm perform better.
To explain why BFS perform worse in that case, I consider that BFS regard the least number of stations is essential. However, it really happens that the less number of stations cause the more distance. As a result, it is necessary for me to find a best algorithm based on the two algorithms.

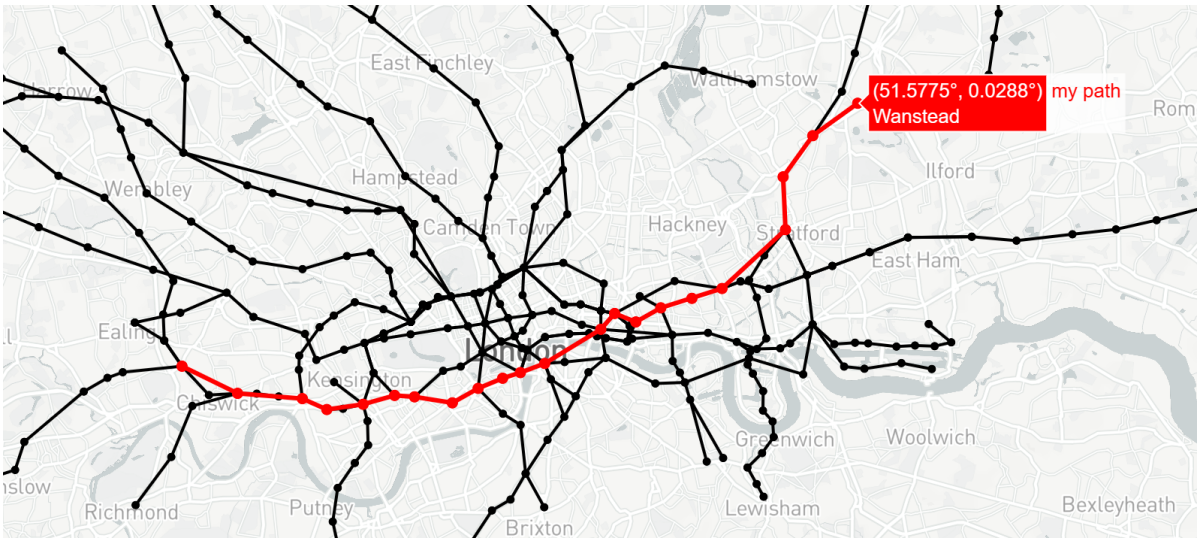
Adjust BFS and A* algorithms

After this motivation, I try my best to promote the two algorithms, which are adjustBFS and adjustAStar function respectively.

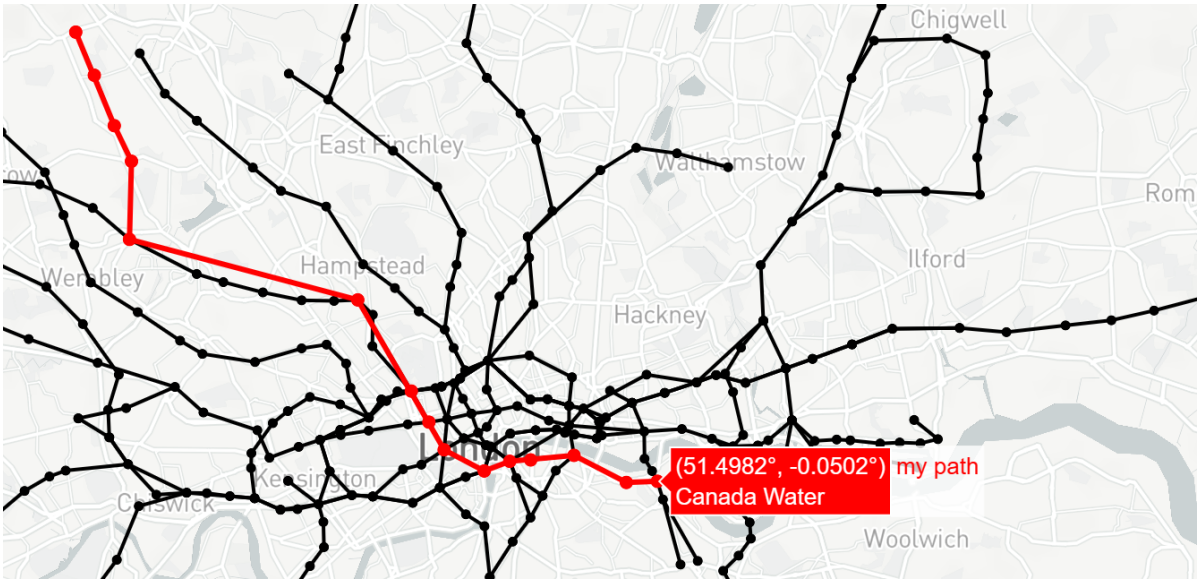
- For **adjustBFS**, I create a dict to store every possible correct path as the key and the distance as the value. When the algorithm finds the first path, I **continue** to find another one** instead of** stoping searching. After collecting several paths(the number of paths is a hyperparameter, which is up to the specific case), we **break** the circulation and pick the path which possess the **least** distance as the output.
- For **adjustAStar**, it seems like a trick. I define a hyperparameter to restrain the search process. when the search move to a specific circumstance, I adjust the **_lt_** function in the class Node to make the algorithm take more importance on the **cost** than the heuristic, which shows the distance between current station and the end station.

After numeric tests, the output of adjustBFS and adjustAStar is almost identical. Here are some examples:

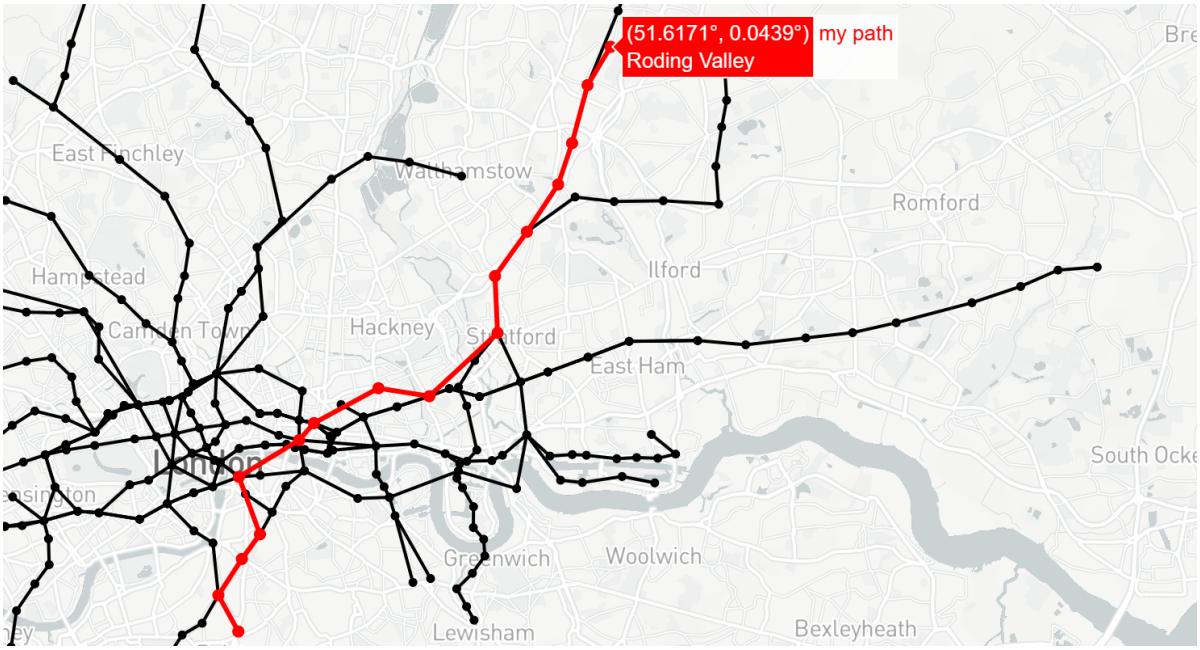
```
python .\find_shortest_path.py "Acton Town" "Wanstead"
```




```
python .\find_shortest_path.py "Stanmore" "Canada Water"
```




```
python .\find_shortest_path.py "Roding Valley" "Brixton"
```



Part 3: Discussion and Analysis

Time Complexity

The time complexity of BFS algorithm is hard to evaluate. Assume every station has at most 5 neighbors, the time complexity may be

$$O(5^n)$$

The time complexity of A* algorithm is

$$O(n * 5^n)$$

- **n** : represent the number of the stations in the path.

Strengths and Limitations

Since I have done plenty of tests on different start_stations and end_stations, I find that the adjustBFS works **better** than the adjustAStar in general. adjustAStar can find more possible paths but cause more time and space. Since the London underground is not too intricate for BFS to search, this brutal algorithm use less time but higher efficiency.
The **limitation** is that the hyperparameter need to adjust to face different circumstances. A unsuitable hyperparameter may cause more running time even wrong answer. So we need to adjust it when obtaining various cases.

Unexpected or Interesting Observations

The A* algorithm may fall into a mistake, so we need the hyperparameter to save it. However, saving too early or too late may both cause errors. Sometimes the result may be riduculous, because it is obviously not the best regulation intuitively.
Another interesting observation was discussed in example 1 before.

Conclusion

The Main Findings and Conclusions

As was discussed before, concentrating on this problem, we can make a definition that the adjustBFS algorithm performs **best**. Since the underground station is not intricate enough for BFS to search paths, I consider that BFS is **more suitable** for this problem, winning for more stability and easier to comprehend.
A* algorithm perform well too. **However**, in practical use, the performance of A* algorithm highly depends on the quality of the chosen heuristic function and the specific circumstances of the problem. I should admit that I haven't found a best heuristic function to make the A* algorithm perform its best.

While A* algorithm can potentially find the shortest path more efficiently, its performance heavily relies on the quality of the heuristic function. When the heuristic function effectively estimates the distance to the goal, A* algorithm generally **outperforms** Breadth-First Search. However, if the heuristic function is not accurate enough, A* algorithm might degrade to performance similar to Breadth-First Search. Like my research, It is no evidence to prove that A* algorithm perform better than BFS algorithm.

Potential Future Improvements or Research Directions

When it comes to the algorithms I designed, I consider it has some potential promotions.

- **heuristic function**: The most suitable heuristic function need to be found in the future.
- **hyperparameters**: I need a automatic way to adjust the hyperparameters with estimating possible paths first, instead of adjusting the hyperparameters when giving a specific case.

The future directions of this research may be numeric. I am interested in adding time into consideration, which is more suitable in reality. The algorithm should show the best time or the best distance or both of them. From my own perspective, the demand for a feasible heuristic function may be beyond imagination.