Autonomous Mobile Robot Homework #3

A0263121W Zeng Jinling A0263252L Shen Xiaoting

1 Introduction to the algorithm implementation

This task involves planning a path between five given key locations on a map of Vivocity level 2. The map is represented as a grid with each grid cell representing a 0.2m* 0.2m square area. The five locations on the map are: start, snacks, store, movie, and food.The goal is to find the most efficient route for a person to visit all four locations and return to the start. To solve this problem, various planning algorithms are used including A*, Dijkstra and Greedy Best First algorithm, and each algorithm has its pros and cons. We implement these algorithms and get the results of planned path, travelled distance, total run time and summarize the trade-offs between different algorithms.

2 Obstacle expansion for pre-processing of a map

According to the requirement of the task, a human footprint should be no less than 0.3m radius of a circular. This means that the algorithm should ensure that the path should allow the human to move through the grid map without encountering any obstacles or becoming stuck in narrow spaces that are too small for their footprint. There are many narrow roads in the original map and we should do obstacles expansion before we do path planning.

We expand the obstacle to next four pixels of up, down, left, and right. Figure 2.1 shows the difference of before and after the obstacle expansion. Some narrow roads disappear and prevent human stuck in these areas.

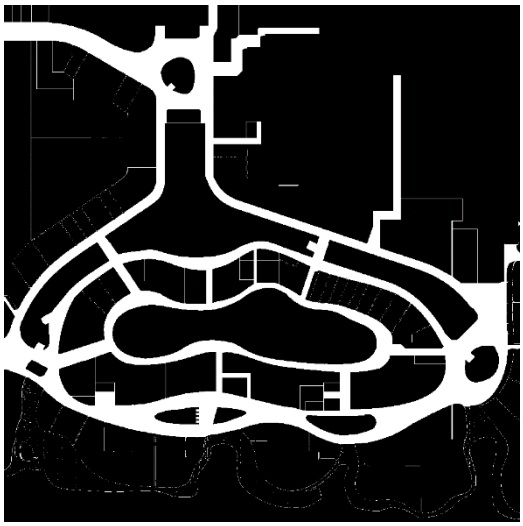

Figure 2.1 Original map

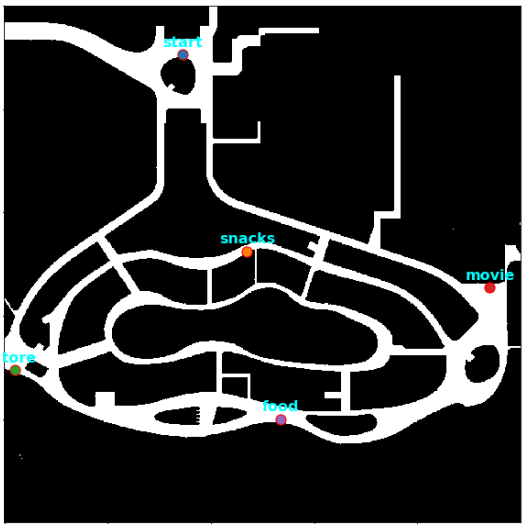

Figure 2.2 Pre-processed map with obstacle expansion

3 Implementation of A*

We implement the A* algorithm to find the shortest path of each location and the pseudocode is shown in the table 3.1

Table 3.1 Pseudocode of algorithm A*

Algorithm A*

```
Input: Grid map, Start Node, Goal Node
Output: Path, distance
Initialize: Open list[start node], close list as a set
        Class Node: x, y, g, h, f, parent, cost
While open list is not empty:
    Current_node = min(open list, node.f)
    if current_node == goal_node:
        return path, distacne
    Close_list.add(current_node)
    Open_list.remove(current_node)
    Get neighbors of current_node (8-conectivity)
    for neighbor in neighbors:
        if grid(neighbor) == 0 or neighbor in closed_list:
            continue
        New_g = current_node.g + cost
        if neighbor not in open_list or new_g < neighbor.g:
            Neighbor.g = new_g
            Neighbor.h = h(neighbor, goal) (Manhattan distance)
            Neighbor.f = g+h
            Neighbor.parent = current_node
            if neighbor not in open_list:
                Open_list.append(neighbor)
```

We can get the results including the distance between location shown in table 3.2 and the visualization routes are shown in figure 3.1. The shorter distance between two destination are highlighted as blue, and the visualization map is only shown the shortest route between locations.

The distance function between two grid cells can be described by different ways like Euclidean distance, Manhattan distance, Chebyshev distance. Then we try different calculation methods to get the best routes between locations. The results are shown in table 3.3 and 3.4, figure 3.2 and 3.3.

(1) Manhattan distance analysis

Table 3.2 Distance between locations with A* algorithm (Manhattan distance)

From / To	**Start**	**Snacks**	**Store**	**Movie**	**Food**
Start	0	147.258	164.036	182.866	233.658
Snacks	144.142	0	118.506	107.966	136.342
Store	159.142	122.42	0	218.640	121.568
Movie	180.570	109.934	223.888	0	117.856
food	232.410	135.194	114.726	117.692	0

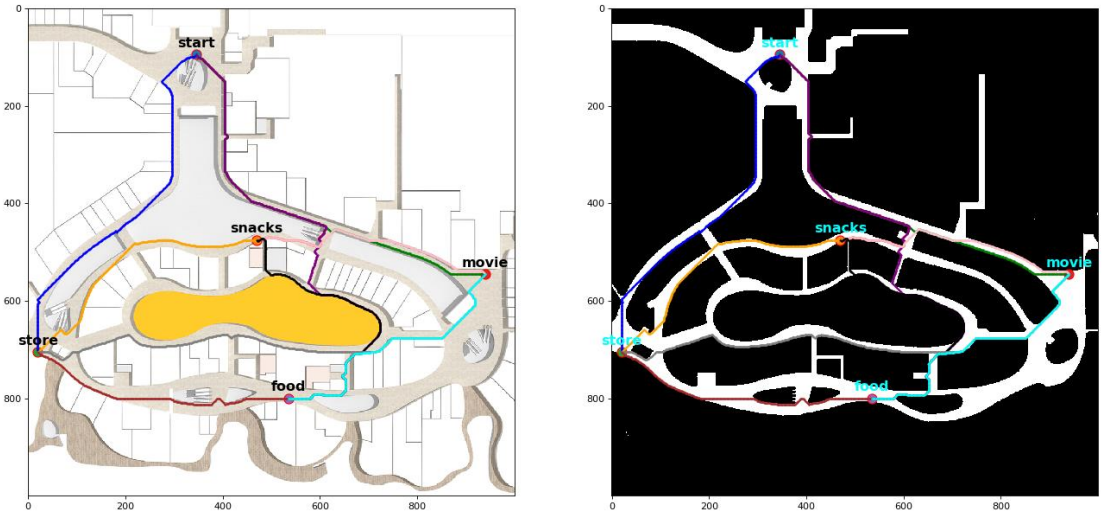

Figure 3.1 Visualization of best routes between locations with A* algorithm (Manhattan distance)

Manhattan distance is restricted to the four cardinal directions because it measures distance by adding up the absolute differences in x and y coordinates separately, without taking into account any diagonal movement. It over-estimates the true distance in this situation that are diagonal movements that allow for shorter travel.

(2) Euclidean distance analysis

Table 3.3 Distance between locations with A* algorithm (Euclidean distance)

From / To	Start	Snacks	Store	Movie	Food
Start	0	144.634	159.654	180.242	227.518
Snacks	142.666	0	115.882	107.146	136.014
Store	158.814	118.648	0	211.752	110.954
Movie	179.422	107.146	212.244	0	116.426
food	227.864	134.702	110.790	116.754	0

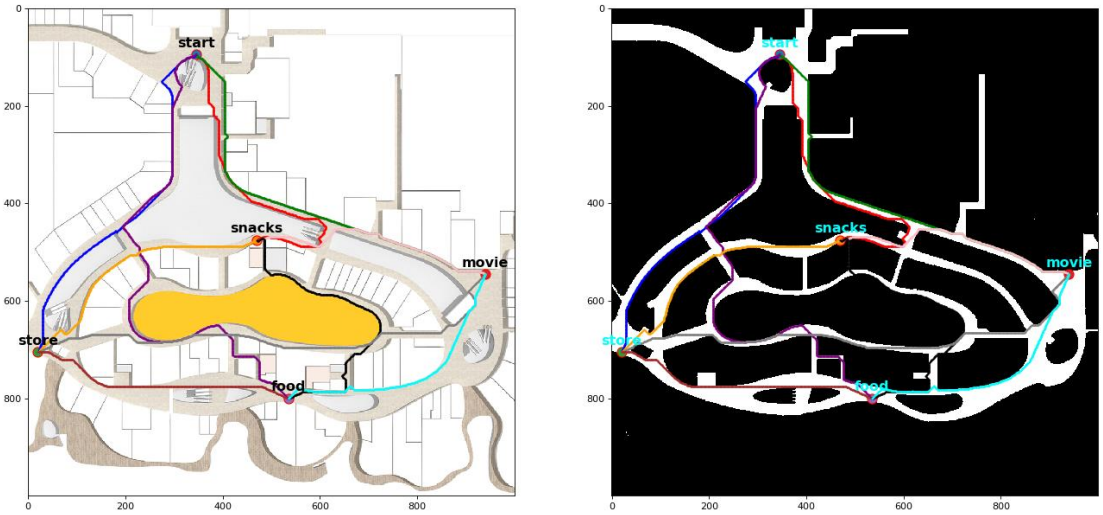

Figure 3.2 Visualization of best routes between locations with A* algorithm (Euclidean distance)

This Euclidean distance is more accurate than Manhattan distance because diagonal

movements are allowed. The shortest distance between location is smaller or close to the Manhattan distance. However, it is more computationally expensive according to the running time of the code shown in table 2.4.

(3) Chebyshev distance analysis

Table 3.4 Distance between locations with A* algorithm (Chebyshev distance)

From / To	Start	Snacks	Store	Movie	Food
Start	0	147.472	158.182	179.094	237.324
Snacks	149.996	0	131.028	113.526	176.714
Store	169.704	115.062	0	228.588	112.876
Movie	194.680	110.214	228.800	0	127.724
food	243.042	178.408	112.560	142.950	0

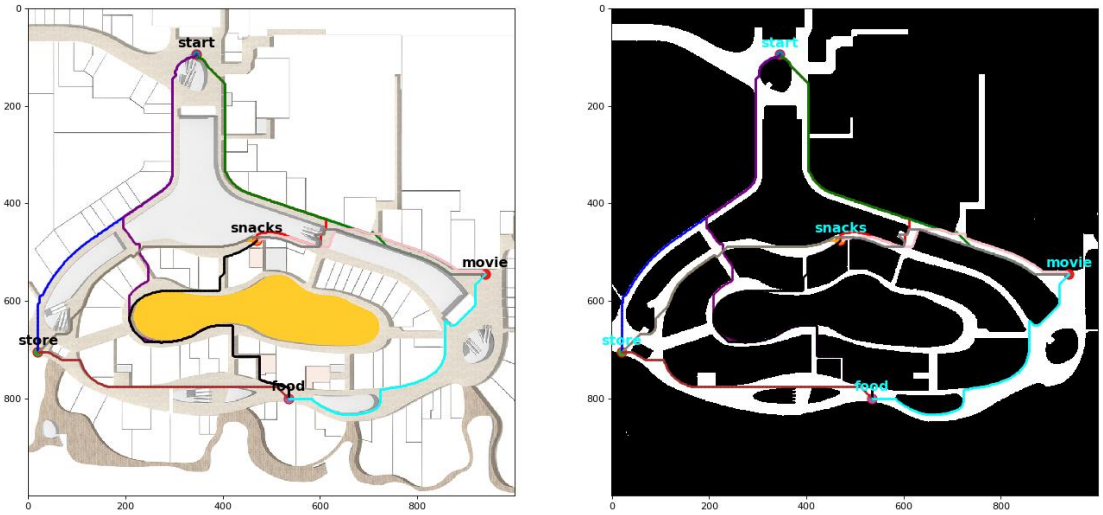

Figure 3.3 Visualization of best routes between locations with A* algorithm (Chebyshev distance)

Chebyshev distance allows the movement in any direction, because it accounts for both horizontal and vertical distance equally. It is much less computationally expensive than Manhattan distance and Euclidean distance according to table 2.4.

Table 3.5 Running time of different heuristic function with A* algorithm and other algorithms

	Manhattan	Euclidean	Chebyshev	Dijkstra	GBFS (Manhattan)
Running time	1min 24.8s	2min 51.9s	33.5s	5min 30.9s	31.8s

(4) Problem of solving the problem

First, we implement the A* algorithm with two list: open_list and close_list. The elements in the list are the grid nodes including the (x, y, g, h, f, parent, cost). If I want to check whether a node has appeared in the open_list or close_list, I have to go over all the nodes in the list and compare their coordinates. The running time of the code is about 20 minutes. Then I change the close_list to be a set structure, then the running time is about 1 minutes. The efficiency has been dramatically improved.

4 Degenerate to Dijkstra and Greedy Best First Search Algorithm

Table 4.1 Distance from locations with Dijkstra algorithm

From / To	Start	Snacks	Store	Movie	Food
Start	0	141.846	154.550	178.438	222.106
Snacks	141.846	0	114.406	107.146	132.570
Store	154.550	114.406	0	209.128	110.790
Movie	178.438	107.146	209.128	0	113.474
food	222.106	132.570	110.790	113.474	0

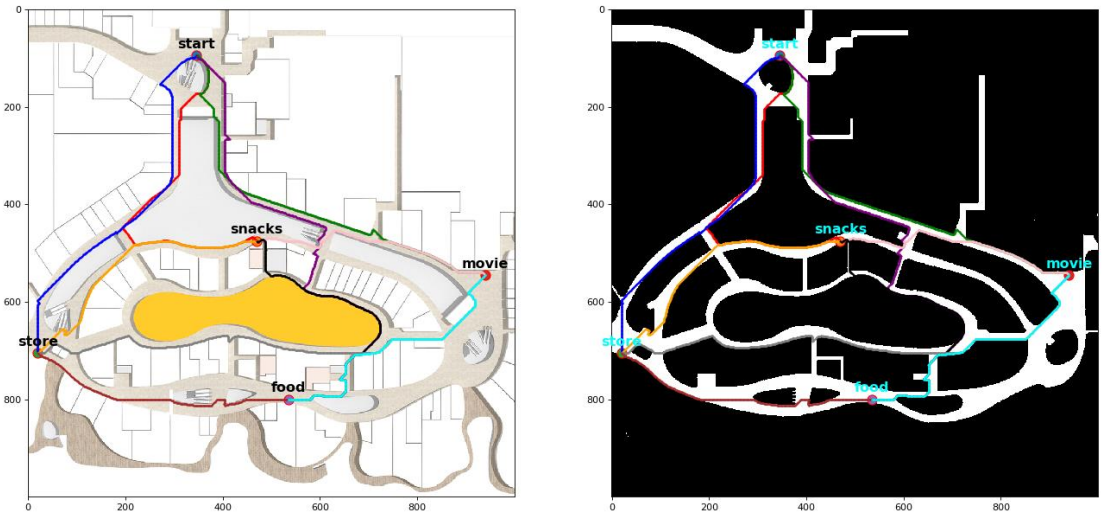

Figure 4.1 Visualization of best routes between locations with Dijkstra algorithm

Dijkstra get the distance between the start and end nodes will be the same regardless of which node is the start and which node is the end. When I exchange the start and end nodes in A* or GBFS, the heuristic function will estimate the distance between the new start node and the goal node differently than it did before. This difference in the estimated distance can cause the algorithm to explore different paths, which can result in a different distance between the start and end nodes.

Table 4.2 Distance from locations with Greedy Best First Search Algorithm

From / To	Start	Snacks	Store	Movie	Food
Start	0	153.638	166.364	184.748	293.390
Snacks	145.906	0	120.834	109.120	182.113
Store	160.342	123.358	0	221.978	132.334
Movie	211.656	142.684	255.742	0	120.538
food	240.338	136.794	115.352	119.692	0

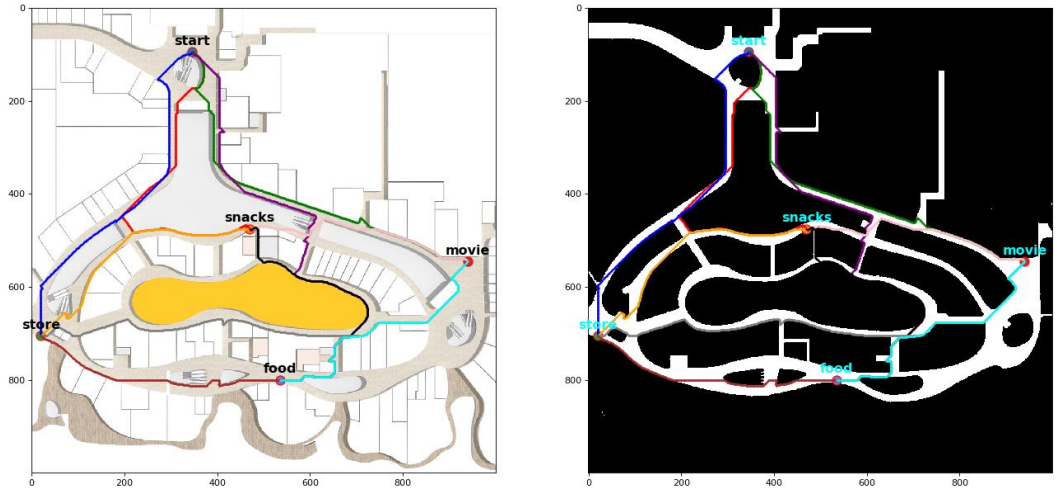

Figure 4.2 Visualization of best routes between locations with Greedy Best First Search Algorithm

GBFS is not guaranteed to find the optimal path because it does not take into account the cost of moving between nodes. The routes it gets are longer than other two algorithms. Greedy Best First Search is very fast with about 30s and requires very little memory, making it a good choice for large-scale environments. Dijkstra is guaranteed to find the optimal path, the best route (shortest distance) is shorter than other algorithms. However, Dijkstra does not use a heuristic function to guide its search, which can lead to slower search times which costs 5.5 min.

5 Travelling shopper problem

(1) TSP with DFS algorithm

We use the results of distance between locations with Dijkstra algorithm and find the optimal route to visit all stores and come back to the start location. There are only 5 locations and we can search every possible location with DFS algorithm. We select the start and there are four rest location and we can choose one of the four rest locations in the first loop. We can choose one of locations for the third visit in the rest three locations. We perform the same loop until there are only one location left. There are 4*3*2 times for the calculation and we get the distance of best route is 627.81m. The visiting order is start→store→food→movie→snacks→start and the route is shown in figure 5.1.

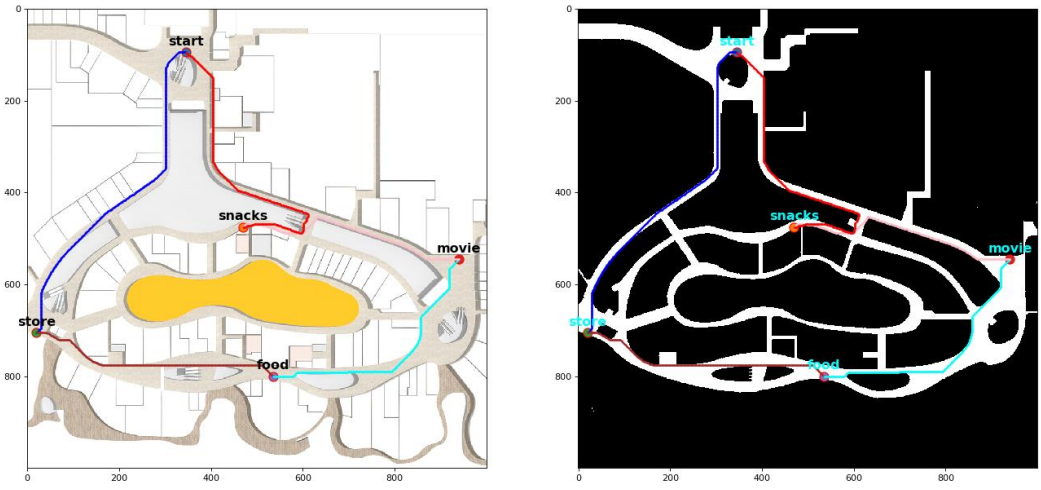

Figure 5.1 Visualization of the shortest routes of visiting all the locations with DFS

(2) TSP with Dynamic Programming algorithm

According to the results of Dijkstra algorithm, we can get the distance matrix which is symmetric to find optimal route. We can form a set including the locations and keep on finding the optimal route with subset. It is clear that this problem can be solved by dynamic programming. The key step to perform dynamic programming is Formulate state and transition relationship. We define a 2 dimension array f[i][j] which means that the shortest route ending at location j and passing all the location in set i. The set i can be shown in binary, and the bits of 1 which means the locations are included in the set (The detailed explanation is shown in the code). We keep on checking whether the location in the set. If it is not in the set, we add it to the set and update the shortest distance to this added position.

After successfully implementation of the DP algorithm, we get the distance of best route is 627.81m. The visiting order is start→store→food→movie→snacks→start and the route is shown in figure 5.2 which is the same of DFS.

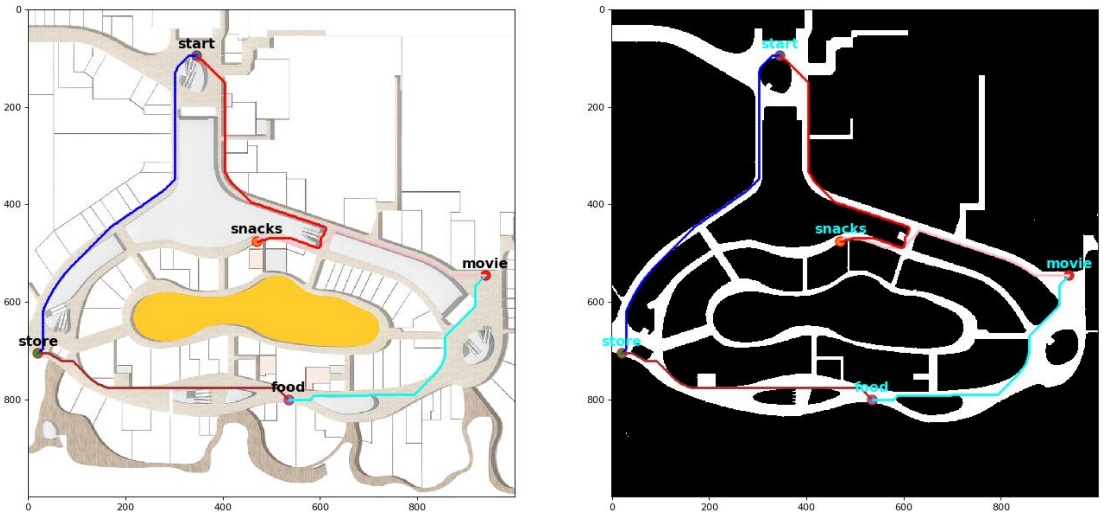

Figure 5.2 Visualization of the shortest routes of visiting all the locations with Dynamic Programming algorithm

(3) Comparison and conclusion

Because of the symmetric distance matrix, the optimal route is same start→store→food→movie→snacks→start and start→snacks→movie→food→store→start. The two algorithms can both get the optimal route with 672.81m and the running time is close to 0.

However, if we have too many places to visit, the time complexity and auxiliary space have a dramatically decrease of dynamic programming compared with DFS. The DFS algorithm is not suitable for the problem with too many locations and will be a huge cost of computer.