National University of Singapore

College of Design and Engineering

ME5413 Autonomous Mobile Robotics: Homework 3

Group 9:
Chen Yihui A0263115N
Wang Renjie A0263387U

Supervisor:
Prof. Marcelo H Ang Jr

Email Address: *e1010473@u.nus.edu*
e1010745@u.nus.edu

March 18, 2023

1 Task 1: Graph Search Algorithms

1.1 Problem statement

In this section, the A* graph search algorithm is used for path planning on the map of Vivocity level 2 (Fig. 1). Firstly, the original floor plan (left) is transformed to a 1000×1000 grayscaled version (right), where the value of each pixel marks if the grid is occupied (255 for free space, 0 for obstacles). The visitors plan to visit the five key locations ("start", "snacks", "store", "movie" and "food") as shown on the map, and optimal path is required to be planned between each pair of those points. In our algorithm implementation, we assume the map resolution is 0.2m×0.2m for each grid and the visitor has a circular footprint of 0.3m radius.

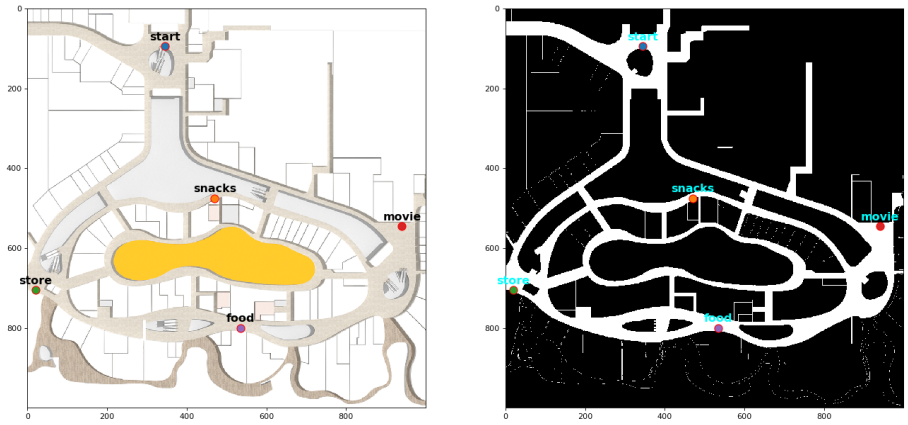


Figure 1: Original and grayscaled map of the Vivocity level 2

1.2 Implement of A* algorithm

Algorithm 1: A* path planning algorithm

Input: start point s_{start}, goal point s_{end}
Output: planned path *path*

Initialize *Openset*, *Closeset*, *g_cost* and *parent_node*;
while *Openset is not empty* **do**
  Pop one node $s_{current}$ from the head of *Openset*;
  Add $s_{current}$ to *Closeset*;
  if $s_{current}$ *is the goal point* s_{end} **then**
    break;
  end
  for *each neighbor node* s_n *of* $s_{current}$ **do**
    Compute the new *g_cost*: $g_{new} = g\_cost[s_{current}] + edgecost(s_{current}, s_n)$;
    if s_n *not in g_cost* **then**
      Initialize *g_cost* for s_n: $g\_cost[s_n] = inf$;
    end
    if $g_{new} < g\_cost[s_n]$ **then**
      Rechoose the parent node: $parent\_node[s_n] = s$;
      Update the *g_cost* for s_n: $g\_cost[s_n] = g_{new}$;
      Compute the f value for s_n: $f(s_n) = g\_cost[s_n] + h(s_n)$;
      Push the node s_n with its f value into *Openset*;
    end
  end
end
Retrieve the final path *path* from s_{end} according to the parrent node dictionary *parrent_node*;

The workflow of A* algorithm is given in Alg. 1. At the beginning (line 1), we initialize one priority

queue *Openset*, which will store the node with the highest f value on the top. For each node s, we compute its f value by:

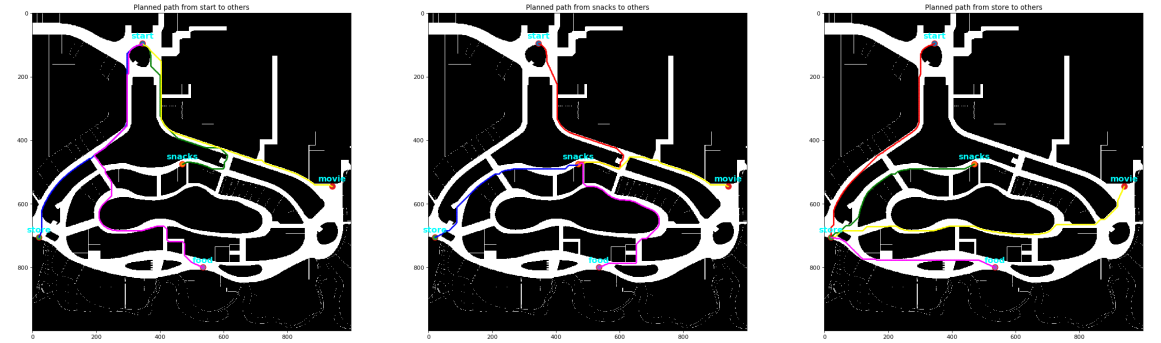
$$f(s) = g(s) + h(s) \tag{1}$$

where $g(s)$ is the path length between s and the start point s_{start}, while $h(s)$ is a heuristic function that measures the distance from s to the goal s_{end}. In the initialization step, we push s_{start} with its f value $f(s_{start}) = h(s_{start})$ into the *Openset*. An empty *Closeset* is also initialized to save the nodes that have been visited. One dictionary *g_cost* is used to store the g value for each node, while *parent_node* dict records the parent of each node. We assume the node has a infinite g value if there's no found path from s_{start} to it, so we initialize $g\_cost[s_{end}] = inf$.

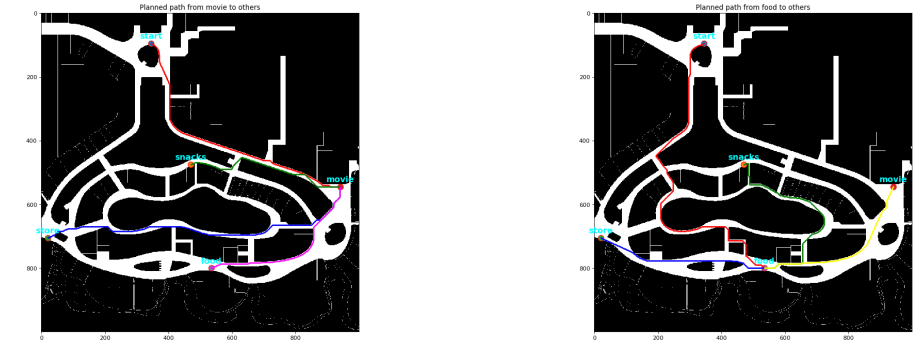
For each current node $s_{current}$, we expand in 8 directions and compute the new g value for each neighbor node s_n (line 9). $edgecost(s_{current}, s_n)$ is measured by the Euclidean distance between the two nodes, and it is set to be inifinite if there's an obstacle. Therefore, collision is checked according to the given footprint radius and map resolution in our algorithm. Then, if the new cost g_{new} is less than its original value (a better path is found for this node), we update g value and rechoose the parent node for it (line 14, 15). After that, the node s_n with its computed f value is pushed into the priority queue *Openset* for future expansion (line 16, 17).

The algorithm ends if the *Openset* is empty (means no more nodes can be expanded and fail to find the path) or we finally expand to the goal point (line 5,6,7). If there exists one way from the start to destination, A* algorithm can always achieve the former case with a shortest path. The length of the optimal path can be got from the dict *g_cost* (i.e. $g\_cost[s_{end}]$), and all visited nodes are saved in *Closeset*. To evaluate our algorithm, we also record the searching time for further comparison.

We first determine the heuristic function as the Euclidean distance between $s_{current}$ and s_{end}. For each key location on the map, we plot the planned paths from it to all other key locations with different color (Fig. 2). Tab. 1 records the shortest distances between each pair of locations, and Tab. 2 & 3 saves the number of visited cells and running time.



(a) From "start" to other key locations (b) From "snacks" to other key locations (c) From "store" to other key locations



(d) From "movie" to other key locations (e) From "food" to other key locations

Figure 2: Path planning using A* algorithm with Euclidean heuristic function

Table 1: Path length between key locations (m)

To \ From	start	snacks	store	movie	food
start	0.00	142.49	155.13	178.89	223.32
snacks	142.49	0.00	114.79	107.51	133.43
store	155.13	114.79	0.00	209.42	110.87
movie	178.89	107.51	209.42	0.00	113.72
food	223.32	133.43	110.87	113.72	0.00

Table 2: Number of visited cells for each pair

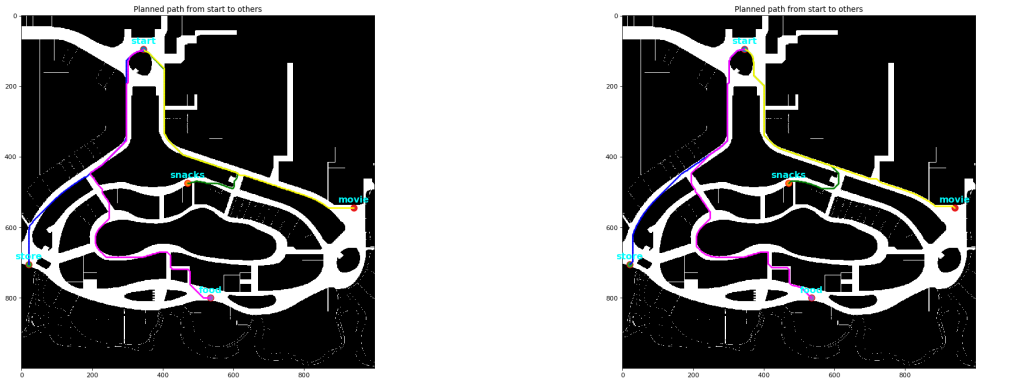
To \ From	start	snacks	store	movie	food
start	0	62604	48950	49582	144731
snacks	36173	0	25162	13943	43622
store	57455	38675	0	89265	18468
movie	42199	14406	52701	0	43275
food	106389	56255	28002	43050	0

Table 3: Running time for each pair (ms)

To \ From	start	snacks	store	movie	food
start	0.00	856.13	440.20	919.54	606.72
snacks	891.98	0.00	205.61	777.32	653.36
store	670.79	876.67	0.00	100.90	243.30
movie	31.48	719.87	949.57	0.00	132.90
food	538.93	747.51	527.95	404.79	0.00

1.3 Comparison of algorithm performance with different heuristic function

We find if the start and goal position are switched, the length of path found by A* algorithm is the same, whereas the path is not definitely the same (see the path from "start" to "snacks" in Fig. 2a and 2b). This is due to the expansion process is not definite, so the number of visited cells as well as the algorithm's running time also changes. To test the performance of A* algorithm with different types of heuristic function, we change the metric to "Manhatan". Also, we degenerate the A* algorithm to Dijkstra's algorithm by setting $h(s) = 0$. We take from "start" to others as an example (all other results are saved in the .ipynb file), and the results are shown in Fig. 3 and Tab. 4. We can see all three algorithms find the optimal path, but Dijkstra's algorithm expands much more nodes and cost much longer time than the other two algorithms, which is because no heuristic function exists to guide it search towards the goal position. Of course, if we only consider the heuristic distance and ignore the g value, A* algorithm will degenerate to Greedy Best First Search algorithm. In that case, the agent can expand quickly towards the goal and find the feasible path, but the path will be far from optimal.



(a) A* (Manhattan heuristic function) (b) Dijkstra's algorithm (without heuristic function)

Figure 3: Performance of algorithm with different heuristic function (from "start" to "snacks")

Table 4: Performance of algorithm with different heuristic functions

Algorithm type	Metrics	snacks	store	movie	food
A* (Euclidean)	Path length (m)	142.49	155.13	178.89	223.32
	Visited cells	62604	48950	49582	144731
	Search time (ms)	856.13	440.20	919.54	606.72
A* (Manhattan)	Path length (m)	142.49	155.13	178.90	223.32
	Visited cells	74603	3304	2801	128348
	Search time (ms)	620.63	367.57	63.91	669.03
Dijkstra's	Path length (m)	142.49	155.13	178.89	223.32
	Visited cells	725364	908754	1309916	2322154
	Search time (ms)	4914.73	5521.74	10592.66	21346.07

We also find the Manhattan heuristic function can almost find the optimal path with less time and visited nodes than using Euclidean distance. However, using heuristic function with Manhattan metrics cannot always find the shortest path, since A* algorithm ensures optimality only when $h(s) \leq h^*(s)$, where $h^*(s)$ represents the real distance from node s to goal position s_{end} (easy to certify $h(s) > h^*(s)$ when using Manhattan distance).

1.4 Possible improvement methods

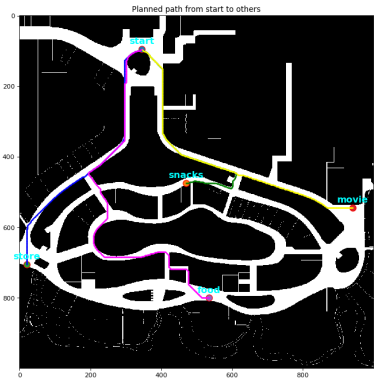
Actually, if we determine our expanding rule as 8-neighbor search, we can compute real distance h^* in closed form (also known as Diagonal heuristic function):

$$h(s) = (|x_f - x_s| + |y_f - y_s| + (\sqrt{2} - 2)min\{|x_f - x_s|, |y_f - y_s|\}) \tag{2}$$

Another technique to reduce the visited nodes is to introduce a tie-breaker, which can break the symmetry of expanding directions. The modified heuristic function becomes:

$$h(s) := h(s) \cdot (1 + p) \tag{3}$$

where p is relevant to the map scale and often set as an extremely small number (0.0001 here). The algorithm performance is shown in Fig. 4. We find the number of visited cells is reduced in two cases, and the search time is much less than that using Euclidean distance without tie-breaker.



Metrics	snacks	store	movie	food
Path length (m)	142.49	155.13	178.89	223.32
Visited cells	76363	35310	47580	154767
Search time (ms)	185.549	31.322	292.285	483.445

Figure 4: Performance of A* algorithm after improvements

2 Task 3: The "Travelling Shopper" Problem

Given Tab. 1, we are required to compute the optimal route to visit all stores and come back to the start location, which is also known as "Travelling Shopper Problem". The simplest way is to enumerate all possible routes and compare their total distance. The time complexity is $O(n!)$ and it can solve the problem quickly in our case since there are only 5 key locations. However, it will become extremely slow when more locations are added.

Another common method is model it as a dynamic programming problem. We define $d(i, V')$ as the lowest cost when the shopper reaches node i and have been to all the nodes in set V'. We mark the start point as '0', so the initial condition becomes:

$$d(0, \{0\}) = 0 \tag{4}$$

When we decide to arrive location 'i' with the lowest cost, we need to compute according to the distance matrix in Tab. 1 and the current state. Then we derive the transition equation:

$$d(i, V' + \{i\}) = \min_{k \in V'} \{d(k, V') + c_{ki}\} \tag{5}$$

After constructing the dynamic programming array, the total distance after coming back to the start point '0' can be obtained by:

$$d\_total = \min_{i \in S} \{d(i, S) + c_{i0}\} \tag{6}$$

where S is the set including all locations (i.e. the shopper man has been to all locations). In our implementation, the set V' has 2^5 possible values and that for i (the current location) is 5, so we construct the dynamic programming array in a shape of $[2^5, 5]$. One technique is to use a binary data to represent the set V', which can facilitate the programming (for example, '11011' represent the set {0,1,3,4}). After filling the array according the initial condition in Eq. 4 and transition rule in Eq. 5, we can directly get the total length by Eq. 6 and also easily obtain the route by retrieving the path from end to start.

We compare the violent enumeration and dynamic programming method by the total length, the whole route and time cost. As is shown in Fig. 5, the total length computed and optimal route obtained are both the same (the route order is just inversed). Surprisingly, in this case the violent enumeration method cost less time than dynamic programming method, since there are only 5 locations here. Actually, the time complexity for the DP method is $O(2^n n^2)$, so when n becomes larger, the DP method can work much more efficiently than the simple enumeration method. We plot the obtained optimal route in Fig. 6.

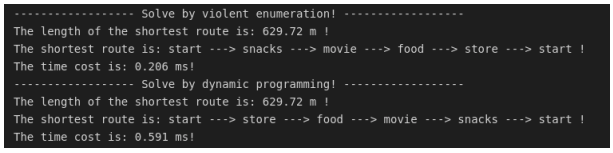


Figure 5: Comparison between the two methods

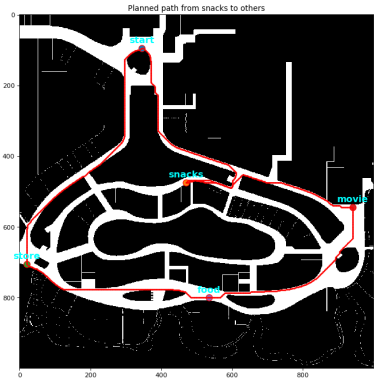


Figure 6: The optimal route obtained by DP method