

Homework #1

Masafumi Endo, M.S. Student in Robotics
ROB534 - Sequential Decision Making in Robotics
OREGON STATE UNIVERSITY

January 27, 2021

Questions

a

i

The cost-to-come function is shown in Table 1. Note that ∞ expresses the cost when the vertices cannot be visited.

Table 1: Cost-to-come function

	a	b	c	d	e
G^*_5	∞	∞	∞	∞	0
G^*_4	∞	∞	7	1	∞
G^*_3	∞	11	4	2	∞
G^*_2	13	8	5	3	∞
G^*_1	10	9	6	6	∞

ii

The cost-to-go function is shown in Table 2. Note that ∞ expresses the cost when the vertices cannot be visited.

Table 2: Cost-to-go function

	a	b	c	d	e
C^*_1	0	∞	∞	∞	∞
C^*_2	∞	2	∞	∞	∞
C^*_3	3	∞	6	∞	∞
C^*_4	∞	5	∞	9	13
C^*_5	6	∞	9	10	10

b

i Sum of Manhattan distance of all tiles to their goal state

Admissible: It doesn't overestimate the optimal cost to reach the goal state.

ii Sum of Manhattan distance of all tiles to their goal state times 2

Inadmissible: It overestimates the optimal cost to reach the goal state.

iii Sum of Euclidean distance of all tiles to their goal state

Admissible: It doesn't overestimate the optimal cost to reach the goal state.

iv Number of tiles not in their goal state

Admissible: It doesn't overestimate the optimal cost to reach the goal state, while it is least informed.

v Number of moves remaining in optimal solution to reach goal state

Admissible: It is equal to the optimal cost to reach the goal state.

c

Based on the above answers, the heuristics are ordered as $iv < iii < i < v$.

Programming Assignment

Step 1

Fig. 1 shows flowcharts of (a) A* and (b) RRT algorithm. Note that O and C in Fig. 1 (a) express the open list and closed list.

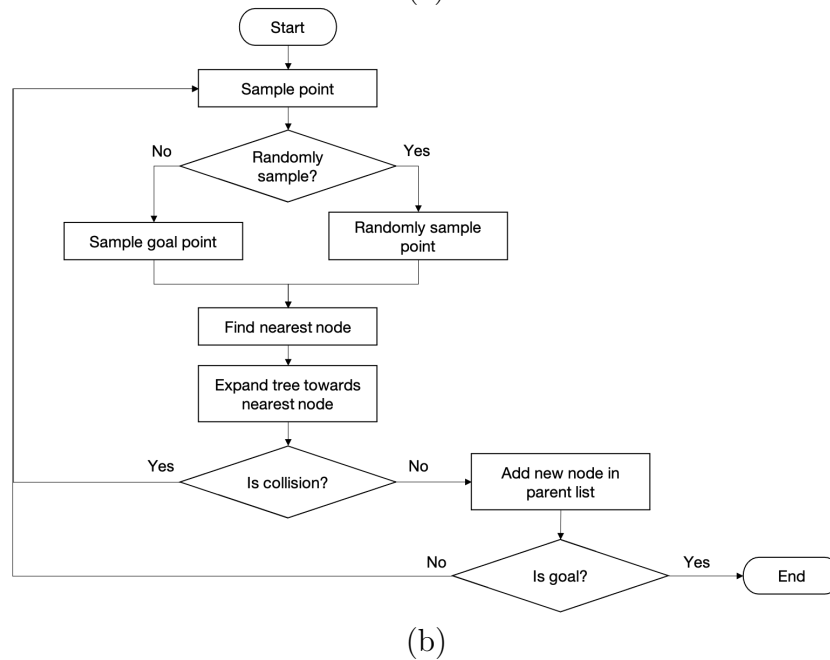
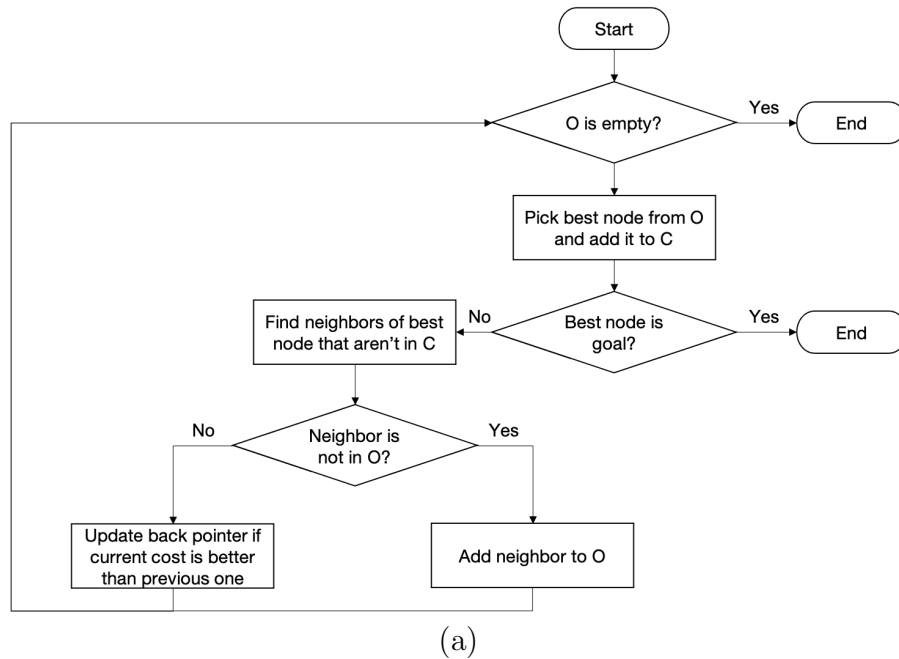


Figure 1: Flowchart of (a) A* and (b) RRT.

Step 2

i

The paths found by A* implementation for 2D space are shown in Fig. 2. I used euclidean distance as admissible heuristic.

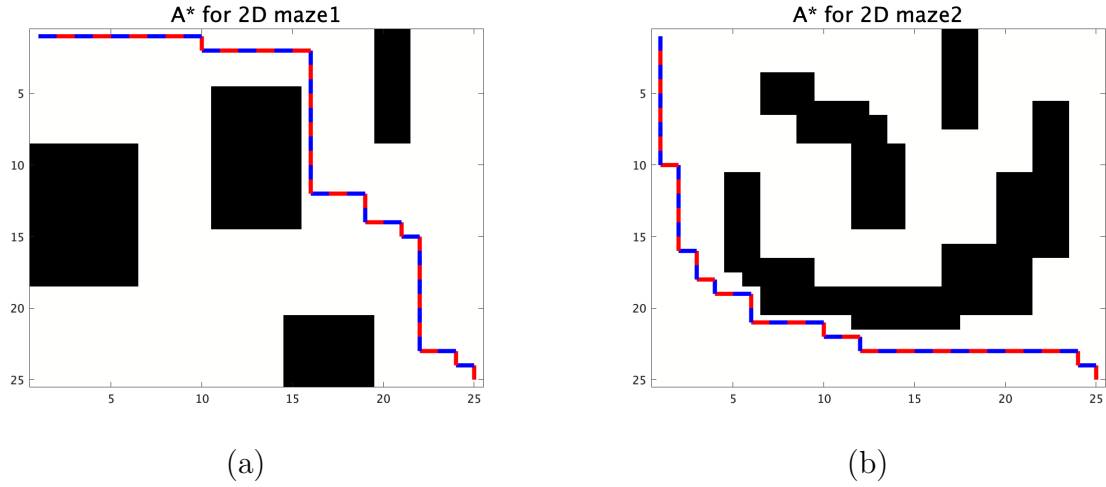


Figure 2: Planned path by A* for 2D maze environments. Note that euclidean distance is used as admissible heuristic.

ii

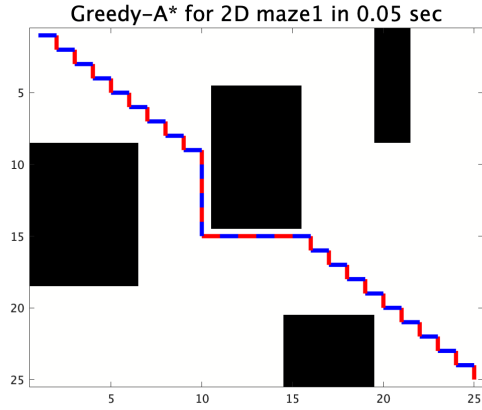
Table 3 and 4 summarize the number of nodes expanded and path length on maze 1 and maze 2 environments for each ϵ value, respectively. In addition, the paths found by greedy-A* implementation for 2D space are shown in Fig. 3. I used euclidean distance as admissible heuristic.

Table 3: Record of ϵ , path length, and number of nodes expanded for maze 1 in 2D space.

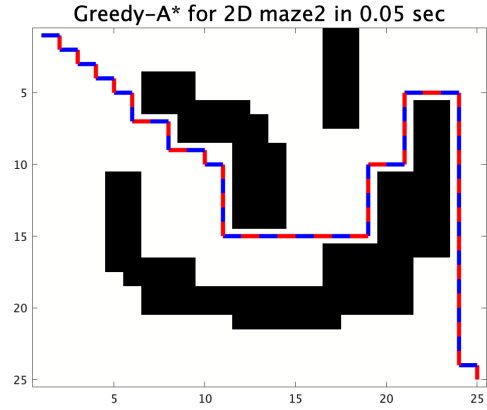
time limit: 0.05 sec			time limit: 0.25 sec			time limit: 1.0 sec		
ϵ	length	nodes	ϵ	length	nodes	ϵ	length	nodes
10	48	49	10	48	49	10	48	49
5.5	48	49	5.5	48	49	5.5	48	49
3.25	48	49	3.25	48	49	3.25	48	49
-	-	-	2.12	48	49	2.12	48	49
-	-	-	1.56	48	50	1.56	48	50
-	-	-	1.28	48	289	1.28	48	289
-	-	-	1.14	48	350	1.14	48	350
-	-	-	1.07	48	369	1.07	48	369
-	-	-	-	-	-	1.03	48	378
-	-	-	-	-	-	1.01	48	379
-	-	-	-	-	-	1.008	48	379
-	-	-	-	-	-	1.004	48	379
-	-	-	-	-	-	1.002	48	379
-	-	-	-	-	-	1.001	48	379
-	-	-	-	-	-	1	48	385

Table 4: Record of ϵ , path length, and number of nodes expanded for maze 2 in 2D space.

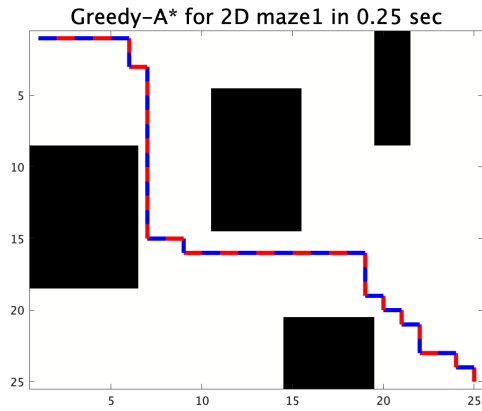
time limit: 0.05 sec			time limit: 0.25 sec			time limit: 1.0 sec		
ϵ	length	nodes	ϵ	length	nodes	ϵ	length	nodes
10	68	155	10	68	155	10	68	155
5.5	68	168	5.5	68	168	5.5	68	168
3.25	52	169	3.25	52	169	3.25	52	169
-	-	-	2.12	48	154	2.12	48	154
-	-	-	1.56	48	284	1.56	48	284
-	-	-	1.28	48	297	1.28	48	297
-	-	-	1.14	48	324	1.14	48	324
-	-	-	1.07	48	351	1.07	48	351
-	-	-	-	-	-	1.03	48	362
-	-	-	-	-	-	1.01	48	363
-	-	-	-	-	-	1.008	48	363
-	-	-	-	-	-	1.004	48	363
-	-	-	-	-	-	1.002	48	363
-	-	-	-	-	-	1.001	48	363
-	-	-	-	-	-	1	48	369



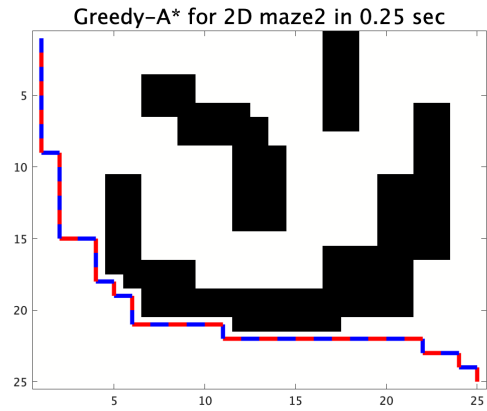
(a)



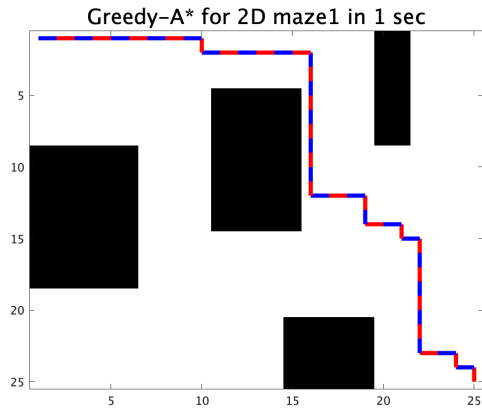
(b)



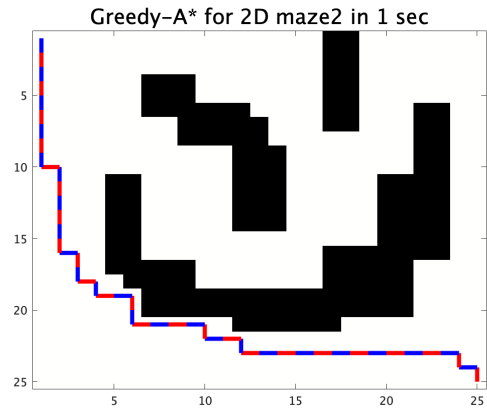
(c)



(d)



(e)



(f)

Figure 3: Planned path by greedy-A* for 2D maze environments. Note that euclidean distance is used as admissible heuristic. (a), (c), and (e) show planned paths for maze1 with time limits 0.05, 0.25, and 1 second. (b), (d), and (f) show planned paths for maze2 with time limits 0.05, 0.25, and 1 second.

iii

The paths found by RRT implementation are shown in Fig. 4. For maze 1 environment, the length of found path is 42 and running time is 0.24 second. For maze 2 environment, the length of found path is 45 and running time is 0.29 second. Note that the length of path and running time may change due to its randomness. Also, the performance will be affected by several parameters: sampling rate, fixed distance to steer towards sampled points, and range to sample random points. I set these parameters as shown in Table 5.

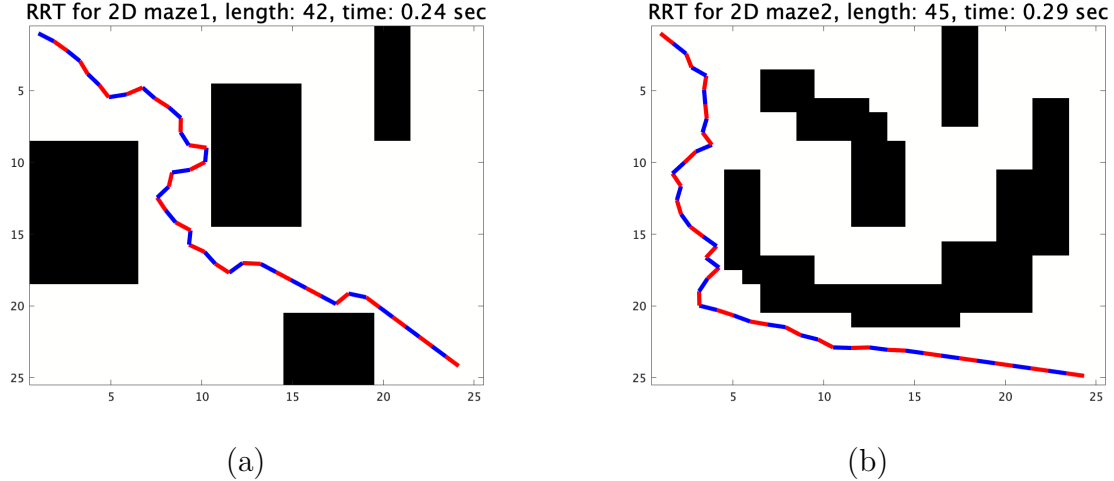


Figure 4: Planned path by RRT for 2D maze environments.

Table 5: User-specified parameters for RRT

sampling rate	0.2
fixed distance	1
sampling range	[0, 25]

Step 3

i

The paths found by A* implementation for 4D space are shown in Fig. 5. I used euclidean distance as admissible heuristic.

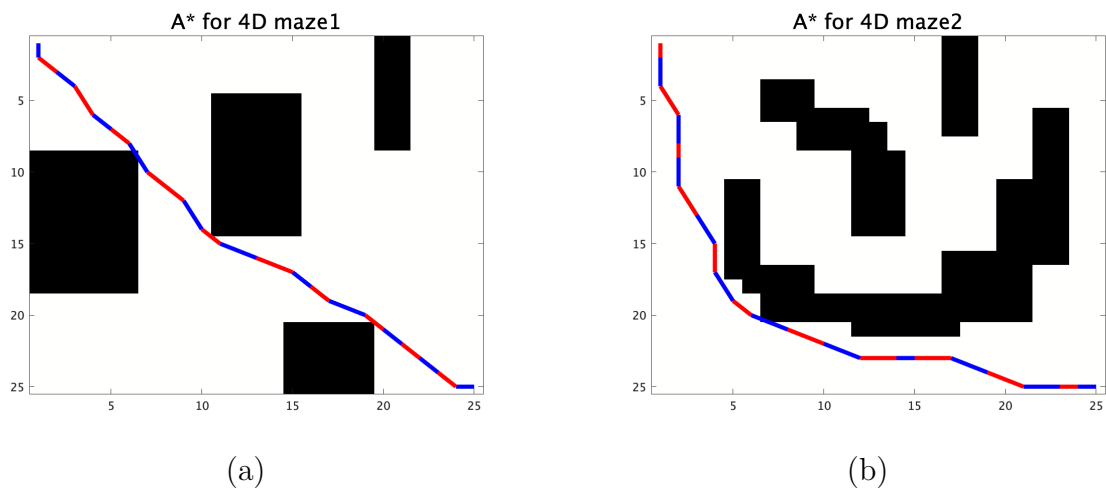


Figure 5: Planned path by A* for 4D maze environments. Note that euclidean distance is used as admissible heuristic.

ii

Table 6 and 7 summarize the number of nodes expanded and path length on maze 1 and maze 2 environments for each ϵ value, respectively. In addition, the paths found by greedy-A* implementation for 4D space are shown in Fig. 6. I used euclidean distance as admissible heuristic.

Table 6: Record of ϵ , path length, and number of nodes expanded for maze 1 in 4D space.

time limit: 0.05 sec			time limit: 0.25 sec			time limit: 1.0 sec		
ϵ	length	nodes	ϵ	length	nodes	ϵ	length	nodes
10	36.6	30	10	36.6	30	10	36.6	30
5.5	36.6	30	5.5	36.6	30	5.5	36.6	30
3.25	36.6	29	3.25	36.6	29	3.25	36.6	29
2.12	36.6	29	2.12	36.6	29	2.12	36.6	29
1.56	36.6	29	1.56	36.6	29	1.56	36.6	29
1.28	36.6	35	1.28	36.6	35	1.28	36.6	35
1.14	36.6	92	1.14	36.6	92	1.14	36.6	92
-	-	-	1.07	35.4	122	1.07	35.4	122
-	-	-	1.03	35.2	176	1.03	35.2	176
-	-	-	1.01	35.2	364	1.01	35.2	364
-	-	-	1.008	35.2	475	1.008	35.2	475
-	-	-	-	-	-	1.004	35.2	528
-	-	-	-	-	-	1.002	35.2	560
-	-	-	-	-	-	1.001	35.2	565
-	-	-	-	-	-	1	35.2	581

Table 7: Record of ϵ , path length, and number of nodes expanded for maze 2 in 4D space.

time limit: 0.05 sec			time limit: 0.25 sec			time limit: 1.0 sec		
ϵ	length	nodes	ϵ	length	nodes	ϵ	length	nodes
10	42.8	523	10	42.8	523	10	42.8	523
-	-	-	5.5	42.4	523	5.5	42.4	523
-	-	-	3.25	42.4	523	3.25	42.4	523
-	-	-	2.12	41.8	537	2.12	41.8	537
-	-	-	-	-	-	1.56	41.8	1193
-	-	-	-	-	-	1.28	41.1	1452
-	-	-	-	-	-	1.14	40.7	1529
-	-	-	-	-	-	1.07	40.5	1557

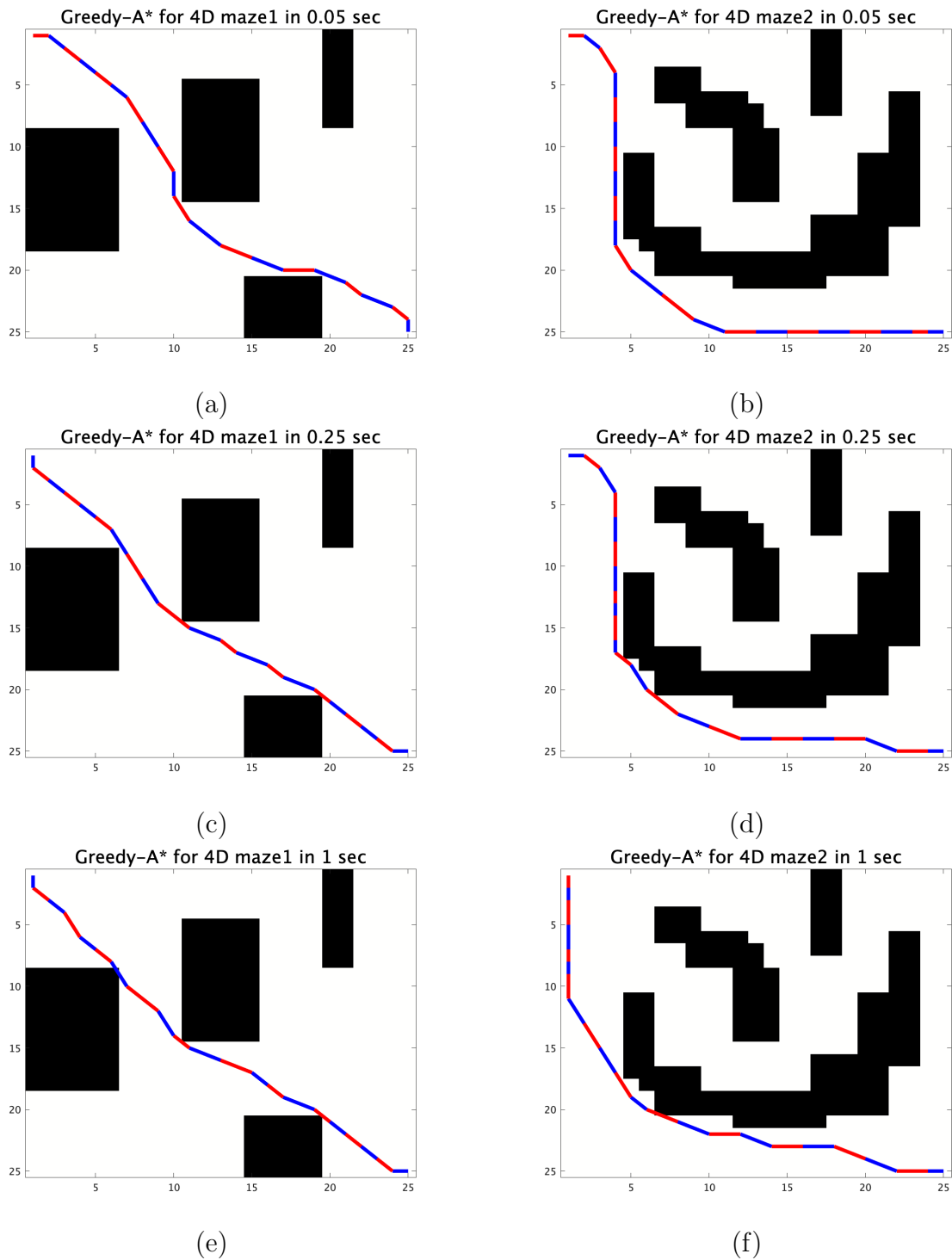


Figure 6: Planned path by greedy-A* for 4D maze environments. Note that euclidean distance is used as admissible heuristic. (a), (c), and (e) show planned paths for maze1 with time limits 0.05, 0.25, and 1 second. (b), (d), and (f) show planned paths for maze2 with time limits 0.05, 0.25, and 1 second.

Discussion Questions

1

The observation is that the larger ϵ causes the less node expansion and longer length of path found by greedy A*. This implies the solution is not optimal when ϵ value is increased. The complexity of maze environment and dimensionality of state space affect the heuristic inflation factor as shown in Table 3, 4, 6, and 7. ϵ depends on heuristic so it is important to first choose the heuristic function then pick ϵ value and deflation rate.

2

The main issue of using RRT to solve the 4D problem is the difficulty of sampling due to the increase of the possible state spaces. To deal with this issue, the more directed sampling strategy would be necessary. In addition, instead of searching state space, RRT can search the path in the configuration space. Searching in C-space would make its random sampling more directed one since it can avoid the increase of the possible state space.

3

The advantage of using A* is that A* can find optimal and complete path if it exist while RRT doesn't guarantee optimality of the path and only guarantee probabilistic completeness. On the other hand, if the size of state space will be increase, A* requires significant computational efforts to find the path. To avoid this issue, we can decrease the amount of discretizing state space. Note that it also sacrifice the path resolution so it is necessary to check whether the path resolution is well enough or not for the robot.

4

A* needs to be modified to re-plan its path online any changes occur in the environment. To do so, it is necessary to use two estimates for its state evaluation: the estimate of cost based on the movement so far and the estimate of cost based on the next motion. To incorporate them for evaluating the priority queue, A* can dynamically modify its path.

5

RRT needs to evaluate its paths with a probabilistic distribution since the situation is partially observable. To evaluate the path towards partially unknown environment, we must estimate the uncertainty and continue to find the path.

Appendix

A* Code for 2D Space

```
1 classdef Astar
2
3     % set properties
4     properties
5         start
6         goal
7         map
8         h_type
9     end
10
11     methods
12         % constructor
13         function obj = Astar(start, goal, map, h_type)
14             obj.start = start;
15             obj.goal = goal;
16             obj.map = map;
17             obj.h_type = h_type;
18         end
19
20         function [path] = search(obj)
21
22             % initialization
23             [~, num_nodes] = get_start(obj.map);
24             priority_list = inf * ones(num_nodes, 1); % list containing priorities
25             open_list = pq_init(1e+4); % open list
26             closed_list = []; % closed list
27             back_pointer = nan * ones(num_nodes, 1); % back-pointer attribute
28             % starting position
29             priority_list(obj.start) = h(obj, obj.start);
30             open_list = pq_set(open_list, obj.start, priority_list(obj.start));
31             back_pointer(obj.start) = -1;
32             while true
33                 % pick best node and remove it from open_list
34                 [open_list, n_best] = pq_pop(open_list);
35                 % add best node to closed_list
36                 closed_list = [closed_list; n_best];
37
38                 % check n_best is goal or not
39                 if n_best == obj.goal
40                     break
41                 end
42
43                 % expand all nodes that are neighbors of n_best
44                 [neighbors, ~] = get_neighbors(obj.map, n_best);
45                 neighbors = neighbors((neighbors == n_best) ~= 1);
46                 for index_neighbor=1:length(neighbors)
47                     neighbor = neighbors(index_neighbor);
48                     % calculate each cost
49                     cost_g = g(obj, n_best, priority_list);
50                     cost_n = cost(obj, n_best, neighbor);
51                     cost_h = h(obj, neighbor);
52                     priority = cost_g + cost_n + cost_h;
53                     % neighbor is in open_list
54                     if pq_test(open_list, neighbor)
55                         if priority < priority_list(neighbor)
56                             % update open_list
57                             priority_list(neighbor) = priority;
58                             open_list = pq_set(open_list, neighbor, priority);
59                             back_pointer(neighbor) = n_best;
60                         end
61                     % neighbor is not in open_list
62                     else
```

```

63         if ismember(neighbor, closed_list)
64             continue
65         % add neighbor to open_list
66         else
67             priority_list(neighbor) = priority;
68             open_list = pq_set(open_list, neighbor, priority);
69             back_pointer(neighbor) = n_best;
70         end
71     end
72 end
73 end
74
75 % after finishing search process, get an optimal path from solution
76 path = get_path(obj, back_pointer);
77 end
78
79 function [path] = get_path(obj, back_pointer)
80 % initialization
81 [x_g, y_g] = state_from_index(obj.map, obj.goal);
82 path = [x_g, y_g];
83 n_best = back_pointer(obj.goal);
84 while true
85     [x_best, y_best] = state_from_index(obj.map, n_best);
86     path = [path; x_best, y_best];
87     n_best = back_pointer(n_best);
88     if n_best == -1
89         break
90     end
91 end
92 end
93
94 % function to calculate actual cost of n_best
95 function [cost_g] = g(obj, n_best, priority_list)
96     cost_g = priority_list(n_best) - h(obj, n_best);
97 end
98
99 % function to calculate actual cost from n_best to neighbor
100 function [cost_n] = cost(obj, n_best, neighbor)
101     [x_best, y_best] = state_from_index(obj.map, n_best);
102     [x_neig, y_neig] = state_from_index(obj.map, neighbor);
103     cost_n = sqrt((x_neig - x_best)^2 + (y_neig - y_best)^2);
104 end
105
106 % function to calculate heuristic cost of the neighbor
107 function [cost_h] = h(obj, neighbor)
108     [x_neig, y_neig] = state_from_index(obj.map, neighbor);
109     [x_goal, y_goal] = state_from_index(obj.map, get_goal(obj.map));
110     if obj.h_type == 'm'
111         % manhattan distance
112         cost_h = abs(x_neig - x_goal) + abs(y_neig - y_goal);
113     elseif obj.h_type == 'e'
114         % euclidian distance
115         cost_h = sqrt((x_neig - x_goal)^2 + (y_neig - y_goal)^2);
116     else
117         disp('select a type of heuristic function!')
118     end
119 end
120 end
121 end

```

Greedy-A* Code for 2D Space

```

1 classdef greedy_Astar
2
3     % set properties

```

```

4     properties
5         start
6         goal
7         map
8         h_type
9         epsilon
10        t_limit
11    end
12
13    methods
14        % constructor
15        function obj = greedy_Astar(start, goal, map, h_type, epsilon, t_limit)
16            obj.start = start;
17            obj.goal = goal;
18            obj.map = map;
19            obj.h_type = h_type;
20            obj.epsilon = epsilon;
21            obj.t_limit = t_limit;
22        end
23
24        function [path] = greedy_search(obj)
25            tic
26            while toc < obj.t_limit
27                [path, nodes_expanded] = search(obj);
28                path_length = get_path_length(path);
29                disp(' ')
30                disp('-----')
31                fprintf('epsilon: %f \n', obj.epsilon)
32                fprintf('number of nodes expanded: %d, path length: %f \n', nodes_expanded,
33                        path_length)
34                disp('-----')
35                disp(' ')
36
37                obj.epsilon = obj.epsilon - 0.5 * (obj.epsilon - 1);
38                if obj.epsilon == 1
39                    break
40                end
41                if obj.epsilon < 1.001
42                    obj.epsilon = 1;
43                end
44            end
45        end
46
47        function [path, nodes_expanded] = search(obj)
48
49            % initialization
50            [~, num_nodes] = get_start(obj.map);
51            priority_list = inf * ones(num_nodes, 1); % list containing priorities
52            open_list = pq_init(1e+4); % open list
53            closed_list = []; % closed list
54            back_pointer = nan * ones(num_nodes, 1); % back-pointer attribute
55            % starting position
56            priority_list(obj.start) = obj.epsilon * h(obj, obj.start);
57            open_list = pq_set(open_list, obj.start, priority_list(obj.start));
58            back_pointer(obj.start) = -1;
59            while true
60                % pick best node and remove it from open_list
61                [open_list, n_best] = pq_pop(open_list);
62                % add best node to closed_list
63                closed_list = [closed_list; n_best];
64
65                % check n_best is goal or not
66                if n_best == obj.goal
67                    break
68                end
69
70                % expand all nodes that are neighbors of n_best
71                [neighbors, ~] = get_neighbors(obj.map, n_best);

```

```

71     neighbors = neighbors((neighbors == n_best) ~= 1);
72     for index_neighbor=1:length(neighbors)
73         neighbor = neighbors(index_neighbor);
74         % calculate each cost
75         cost_g = g(obj, n_best, priority_list);
76         cost_n = cost(obj, n_best, neighbor);
77         cost_h = obj.epsilon * h(obj, neighbor);
78         priority = cost_g + cost_n + cost_h;
79         % neighbor is in open_list
80         if pq_test(open_list, neighbor)
81             if priority < priority_list(neighbor)
82                 % update open_list
83                 priority_list(neighbor) = priority;
84                 open_list = pq_set(open_list, neighbor, priority);
85                 back_pointer(neighbor) = n_best;
86             end
87             % neighbor is not in open_list
88             else
89                 if ismember(neighbor, closed_list)
90                     continue
91                 % add neighbor to open_list
92                 else
93                     priority_list(neighbor) = priority;
94                     open_list = pq_set(open_list, neighbor, priority);
95                     back_pointer(neighbor) = n_best;
96                 end
97             end
98         end
99     end
100
101     % after finishing search process, get an optimal path from solution
102     path = get_path(obj, back_pointer);
103     nodes_expanded = length(closed_list);
104 end
105
106 function [path] = get_path(obj, back_pointer)
107     % initialization
108     [x_g, y_g] = state_from_index(obj.map, obj.goal);
109     path = [x_g, y_g];
110     n_best = back_pointer(obj.goal);
111     while true
112         [x_best, y_best] = state_from_index(obj.map, n_best);
113         path = [path; x_best, y_best];
114         n_best = back_pointer(n_best);
115         if n_best == -1
116             break
117         end
118     end
119 end
120
121 % function to calculate actual cost of n_best
122 function [cost_g] = g(obj, n_best, priority_list)
123     cost_g = priority_list(n_best) - obj.epsilon * h(obj, n_best);
124 end
125
126 % function to calculate actual cost from n_best to neighbor
127 function [cost_n] = cost(obj, n_best, neighbor)
128     [x_best, y_best] = state_from_index(obj.map, n_best);
129     [x_neig, y_neig] = state_from_index(obj.map, neighbor);
130     cost_n = sqrt((x_neig - x_best)^2 + (y_neig - y_best)^2);
131 end
132
133 % function to calculate heuristic cost of the neighbor
134 function [cost_h] = h(obj, neighbor)
135     [x_neig, y_neig] = state_from_index(obj.map, neighbor);
136     [x_goal, y_goal] = state_from_index(obj.map, get_goal(obj.map));
137     if obj.h_type == 'm'
138         % manhattan distance

```

```

139         cost_h = abs(x_neig - x_goal) + abs(y_neig - y_goal);
140     elseif obj.h_type == 'e'
141         % euclidian distance
142         cost_h = sqrt((x_neig - x_goal)^2 + (y_neig - y_goal)^2);
143     else
144         disp('select a type of heuristic function!')
145     end
146 end
147 end
148 end

```

RRT Code for 2D Space

```

1 classdef RRT
2
3     % set properties
4     properties
5         start
6         goal
7         map
8         dist
9         sampling_rate
10        min_rand
11        max_rand
12    end
13
14    methods
15        % constructor
16        function obj = RRT(start, goal, map, dist, sampling_rate, rand_range)
17            obj.start = start;
18            obj.goal = goal;
19            obj.map = map;
20            obj.dist = dist; % dist to steer sampled point
21            obj.sampling_rate = sampling_rate; % rate to sample goal point
22            obj.min_rand = rand_range(1);
23            obj.max_rand = rand_range(2);
24        end
25
26        function [path, runtime] = search(obj)
27
28            % initialization
29            [x_start, y_start] = state_from_index(obj.map, obj.start);
30            [x_goal, y_goal] = state_from_index(obj.map, obj.goal);
31            node_list = [x_start, y_start];
32            parent_list = [nan];
33            node_cnt = 1;
34            tic
35            while true
36                % random sampling
37                [x_rand, y_rand] = get_node(obj);
38
39                % find nearest node
40                index_nearest = knnsearch(node_list, [x_rand, y_rand]);
41                x_near = node_list(index_nearest, 1);
42                y_near = node_list(index_nearest, 2);
43
44                % expand tree towards [x_near, y_near]
45                [x_new, y_new] = steer(obj, [x_near, y_near], [x_rand, y_rand]);
46
47                % check collision
48                dx = x_new - x_near;
49                dy = y_new - y_near;
50                if ~check_hit(obj.map, x_near, y_near, dx, dy)
51                    % add new node
52                    node_list = [node_list; x_new, y_new];

```



```

53         parent_list = [parent_list; index_nearest];
54         node_cnt = node_cnt + 1;
55     end
56
57     % check goal
58     if abs(x_goal - x_new) < 1 && abs(y_goal - y_new) < 1
59         break
60     elseif node_cnt >= 1e+4 % max iteration is 10000
61         disp('error: RRT could not reach goal within 10000 nodes')
62         break
63     end
64 end
65 runtime = toc;
66 % after finishing search process, get a path from solution
67 path = get_path(obj, node_list, parent_list);
68 end
69
70 % function to get node
71 function [x_rand, y_rand] = get_node(obj)
72     if rand(1) > obj.sampling_rate
73         x_rand = (obj.max_rand - obj.min_rand) * rand(1, 1) + obj.min_rand;
74         y_rand = (obj.max_rand - obj.min_rand) * rand(1, 1) + obj.min_rand;
75     else
76         [x_goal, y_goal] = state_from_index(obj.map, obj.goal);
77         x_rand = x_goal;
78         y_rand = y_goal;
79     end
80 end
81
82 % function to steer towards new node
83 function [x_new, y_new] = steer(obj, node_near, node_rand)
84     grad = atan2(node_rand(2) - node_near(2), node_rand(1) - node_near(1));
85     dx = cos(grad) * obj.dist;
86     dy = sin(grad) * obj.dist;
87     x_new = node_near(1) + dx;
88     y_new = node_near(2) + dy;
89 end
90
91 % function to get a final path from goal to start
92 function [path] = get_path(obj, node_list, parent_list)
93
94     % initialization
95     path = [];
96     index_node = length(node_list);
97     while true
98         path = [path; node_list(index_node, :)];
99         index_node = parent_list(index_node, :);
100         if isnan(index_node)
101             break
102         end
103     end
104 end
105 end
106 end

```

A* Code for 4D Space

```

1 classdef Astar_dynamic
2
3     % set properties
4     properties
5         start
6         goal
7         map
8         h_type

```

```

9     end
10
11     methods
12         % constructor
13         function obj = Astar_dynamic(start, goal, map, h_type)
14             obj.start = start;
15             obj.goal = goal;
16             obj.map = map;
17             obj.h_type = h_type;
18         end
19
20         function [path] = search(obj)
21
22             % initialization
23             [~, num_nodes] = get_start_dynamic(obj.map);
24             priority_list = inf * ones(num_nodes, 1); % list containing priorities
25             open_list = pq_init(1e+4); % open list
26             closed_list = []; % closed list
27             back_pointer = nan * ones(num_nodes, 1); % back-pointer attribute
28             % starting position
29             priority_list(obj.start) = h(obj, obj.start);
30             open_list = pq_set(open_list, obj.start, priority_list(obj.start));
31             back_pointer(obj.start) = -1;
32             while true
33                 % pick best node and remove it from open_list
34                 [open_list, n_best] = pq_pop(open_list);
35                 % add best node to closed_list
36                 closed_list = [closed_list; n_best];
37
38                 % check n_best is goal or not
39                 if n_best == obj.goal
40                     break
41                 end
42
43                 % expand all nodes that are neighbors of n_best
44                 [neighbors, ~] = get_neighbors_dynamic(obj.map, n_best);
45                 neighbors = neighbors((neighbors == n_best) ~= 1);
46                 for index_neighbor=1:length(neighbors)
47                     neighbor = neighbors(index_neighbor);
48                     % calculate each cost
49                     cost_g = g(obj, n_best, priority_list);
50                     cost_n = cost(obj, n_best, neighbor);
51                     cost_h = h(obj, neighbor);
52                     priority = cost_g + cost_n + cost_h;
53                     % neighbor is in open_list
54                     if pq_test(open_list, neighbor)
55                         if priority < priority_list(neighbor)
56                             % update open_list
57                             priority_list(neighbor) = priority;
58                             open_list = pq_set(open_list, neighbor, priority);
59                             back_pointer(neighbor) = n_best;
60                         end
61                     % neighbor is not in open_list
62                     else
63                         if ismember(neighbor, closed_list)
64                             continue
65                         % add neighbor to open_list
66                         else
67                             priority_list(neighbor) = priority;
68                             open_list = pq_set(open_list, neighbor, priority);
69                             back_pointer(neighbor) = n_best;
70                         end
71                     end
72                 end
73             end
74
75             % after finishing search process, get an optimal path from solution
76             path = get_path(obj, back_pointer);

```

```

77     end
78
79     function [path] = get_path(obj, back_pointer)
80         % initialization
81         [x_g, y_g, ~, ~] = dynamic_state_from_index(obj.map, obj.goal);
82         path = [x_g, y_g];
83         n_best = back_pointer(obj.goal);
84         while true
85             [x_best, y_best, ~, ~] = dynamic_state_from_index(obj.map, n_best);
86             path = [path; x_best, y_best];
87             n_best = back_pointer(n_best);
88             if n_best == -1
89                 break
90             end
91         end
92     end
93
94     % function to calculate actual cost of n_best
95     function [cost_g] = g(obj, n_best, priority_list)
96         cost_g = priority_list(n_best) - h(obj, n_best);
97     end
98
99     % function to calculate actual cost from n_best to neighbor
100    function [cost_n] = cost(obj, n_best, neighbor)
101        [x_best, y_best, ~, ~] = dynamic_state_from_index(obj.map, n_best);
102        [x_neig, y_neig, ~, ~] = dynamic_state_from_index(obj.map, neighbor);
103        cost_n = sqrt((x_neig - x_best)^2 + (y_neig - y_best)^2);
104    end
105
106    % function to calculate heuristic cost of the neighbor
107    function [cost_h] = h(obj, neighbor)
108        [x_neig, y_neig, ~, ~] = dynamic_state_from_index(obj.map, neighbor);
109        [x_goal, y_goal, ~, ~] = dynamic_state_from_index(obj.map, get_goal(obj.map));
110        if obj.h_type == 'm'
111            % manhattan distance
112            cost_h = abs(x_neig - x_goal) + abs(y_neig - y_goal);
113        elseif obj.h_type == 'e'
114            % euclidian distance
115            cost_h = sqrt((x_neig - x_goal)^2 + (y_neig - y_goal)^2);
116        else
117            disp('select a type of heuristic function!')
118        end
119    end
120 end
121 end

```

Greedy-A* Code for 4D Space

```

1  classdef greedy_Astar_dynamic
2
3      % set properties
4      properties
5          start
6          goal
7          map
8          h_type
9          epsilon
10         t_limit
11     end
12
13     methods
14         % constructor
15         function obj = greedy_Astar_dynamic(start, goal, map, h_type, epsilon, t_limit)
16             obj.start = start;
17             obj.goal = goal;

```

```

18     obj.map = map;
19     obj.h_type = h_type;
20     obj.epsilon = epsilon;
21     obj.t_limit = t_limit;
22 end
23
24 function [path] = greedy_search(obj)
25     tic
26     while toc < obj.t_limit
27         [path, nodes_expanded] = search(obj);
28         path_length = get_path_length(path);
29         disp(' ')
30         disp('-----')
31         fprintf('epsilon: %f \n', obj.epsilon)
32         fprintf('number of nodes expanded: %d, path length: %f \n', nodes_expanded,
33             path_length)
34         disp('-----')
35         disp(' ')
36
37         obj.epsilon = obj.epsilon - 0.5 * (obj.epsilon - 1);
38         if obj.epsilon == 1
39             break
40         end
41         if obj.epsilon < 1.001
42             obj.epsilon = 1;
43         end
44     end
45 end
46
47 function [path, nodes_expanded] = search(obj)
48
49     % initialization
50     [~, num_nodes] = get_start_dynamic(obj.map);
51     priority_list = inf * ones(num_nodes, 1); % list containing priorities
52     open_list = pq_init(1e+4); % open list
53     closed_list = []; % closed list
54     back_pointer = nan * ones(num_nodes, 1); % back-pointer attribute
55
56     % starting position
57     priority_list(obj.start) = obj.epsilon * h(obj, obj.start);
58     open_list = pq_set(open_list, obj.start, priority_list(obj.start));
59     back_pointer(obj.start) = -1;
60
61     while true
62         % pick best node and remove it from open_list
63         [open_list, n_best] = pq_pop(open_list);
64         % add best node to closed_list
65         closed_list = [closed_list; n_best];
66
67         % check n_best is goal or not
68         if n_best == obj.goal
69             break
70         end
71
72         % expand all nodes that are neighbors of n_best
73         [neighbors, ~] = get_neighbors_dynamic(obj.map, n_best);
74         neighbors = neighbors((neighbors == n_best) ~= 1);
75         for index_neighbor=1:length(neighbors)
76             neighbor = neighbors(index_neighbor);
77             % calculate each cost
78             cost_g = g(obj, n_best, priority_list);
79             cost_n = cost(obj, n_best, neighbor);
80             cost_h = obj.epsilon * h(obj, neighbor);
81             priority = cost_g + cost_n + cost_h;
82             % neighbor is in open_list
83             if pq_test(open_list, neighbor)
84                 if priority < priority_list(neighbor)
85                     % update open_list
86                     priority_list(neighbor) = priority;
87                     open_list = pq_set(open_list, neighbor, priority);
88                 end
89             end
90         end
91     end
92 end

```

```

85         back_pointer(neighbor) = n_best;
86     end
87     % neighbor is not in open_list
88     else
89         if ismember(neighbor, closed_list)
90             continue
91         % add neighbor to open_list
92     else
93         priority_list(neighbor) = priority;
94         open_list = pq_set(open_list, neighbor, priority);
95         back_pointer(neighbor) = n_best;
96     end
97 end
98 end
99 end
100
101 % after finishing search process, get an optimal path from solution
102 path = get_path(obj, back_pointer);
103 nodes_expanded = length(closed_list);
104 end
105
106 function [path] = get_path(obj, back_pointer)
107 % initialization
108 [x_g, y_g, ~, ~] = dynamic_state_from_index(obj.map, obj.goal);
109 path = [x_g, y_g];
110 n_best = back_pointer(obj.goal);
111 while true
112     [x_best, y_best, ~, ~] = dynamic_state_from_index(obj.map, n_best);
113     path = [path; x_best, y_best];
114     n_best = back_pointer(n_best);
115     if n_best == -1
116         break
117     end
118 end
119 end
120
121 % function to calculate actual cost of n_best
122 function [cost_g] = g(obj, n_best, priority_list)
123     cost_g = priority_list(n_best) - obj.epsilon * h(obj, n_best);
124 end
125
126 % function to calculate actual cost from n_best to neighbor
127 function [cost_n] = cost(obj, n_best, neighbor)
128     [x_best, y_best, ~, ~] = dynamic_state_from_index(obj.map, n_best);
129     [x_neig, y_neig, ~, ~] = dynamic_state_from_index(obj.map, neighbor);
130     cost_n = sqrt((x_neig - x_best)^2 + (y_neig - y_best)^2);
131 end
132
133 % function to calculate heuristic cost of the neighbor
134 function [cost_h] = h(obj, neighbor)
135     [x_neig, y_neig, ~, ~] = dynamic_state_from_index(obj.map, neighbor);
136     [x_goal, y_goal, ~, ~] = dynamic_state_from_index(obj.map, get_goal(obj.map));
137     if obj.h_type == 'm'
138         % manhattan distance
139         cost_h = abs(x_neig - x_goal) + abs(y_neig - y_goal);
140     elseif obj.h_type == 'e'
141         % euclidian distance
142         cost_h = sqrt((x_neig - x_goal)^2 + (y_neig - y_goal)^2);
143     else
144         disp('select a type of heuristic function!')
145     end
146 end
147 end
148 end

```