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.

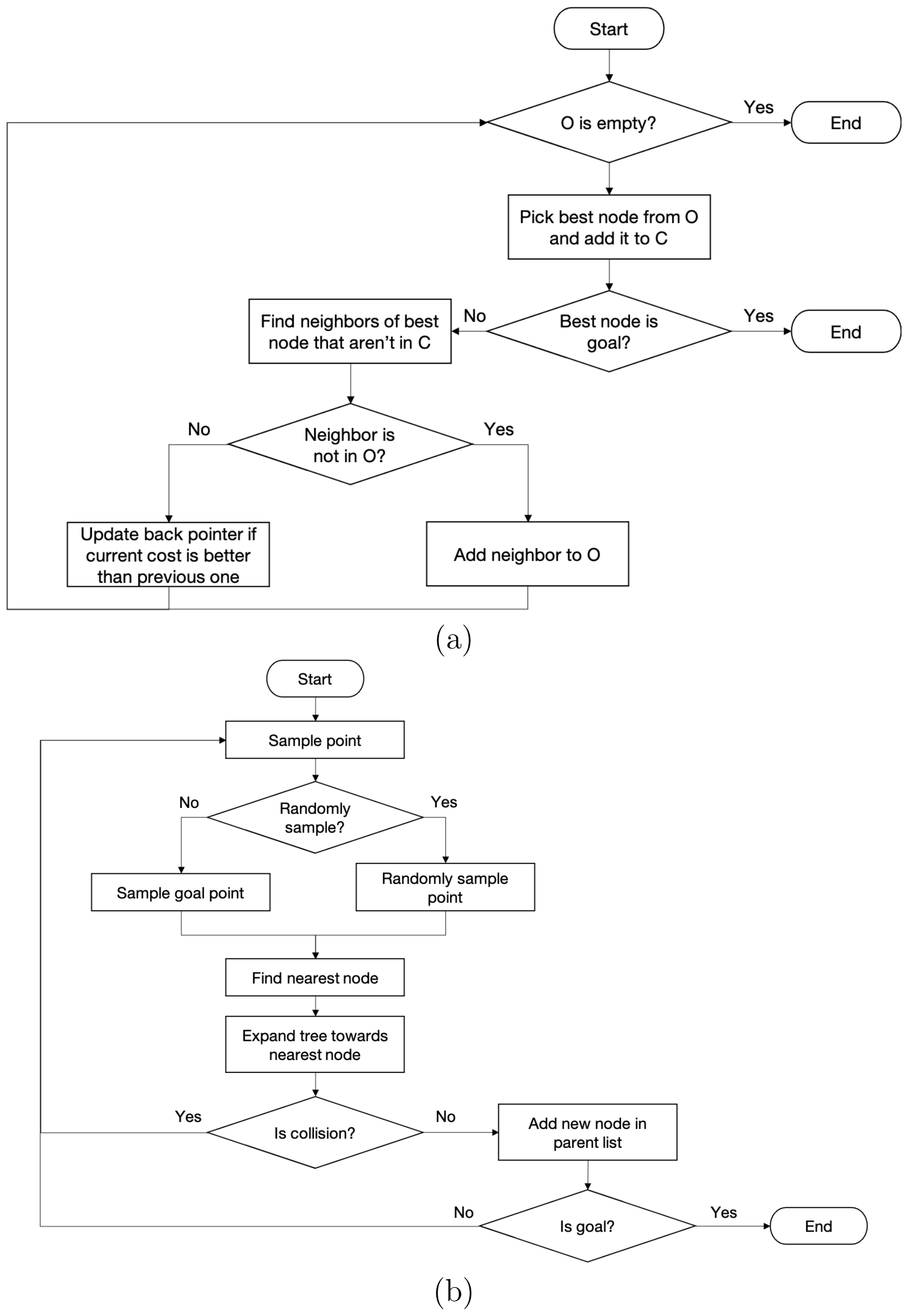
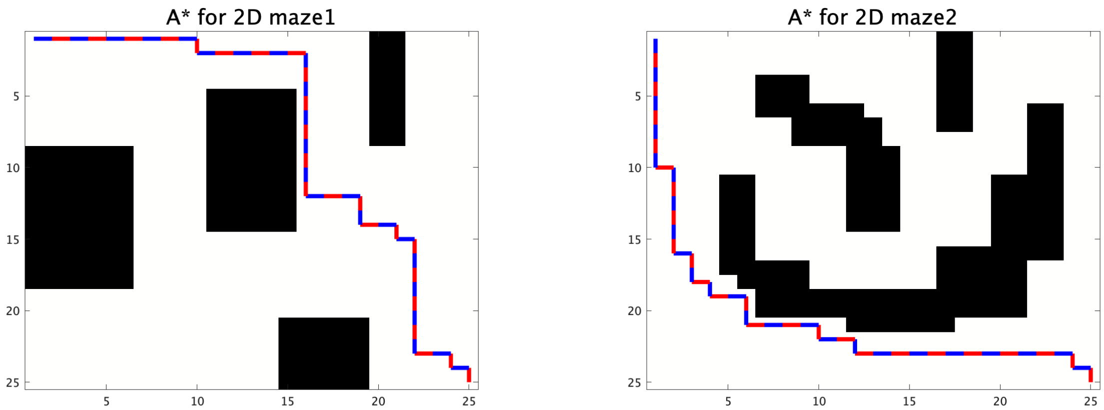


Figure 1: Flowchart of (a) A* and (b) RRT.

Step 2

i

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



(a) (b)

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

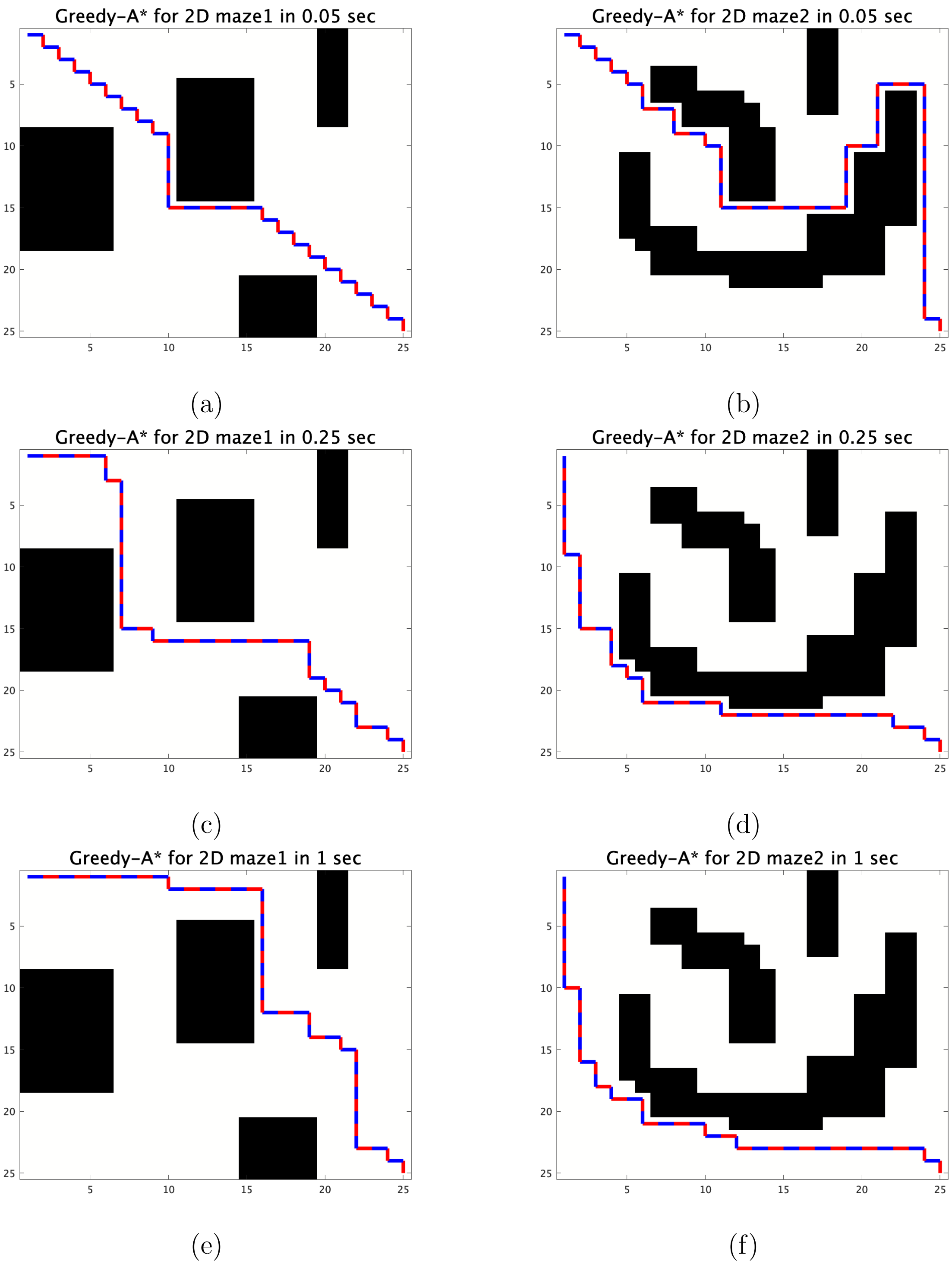
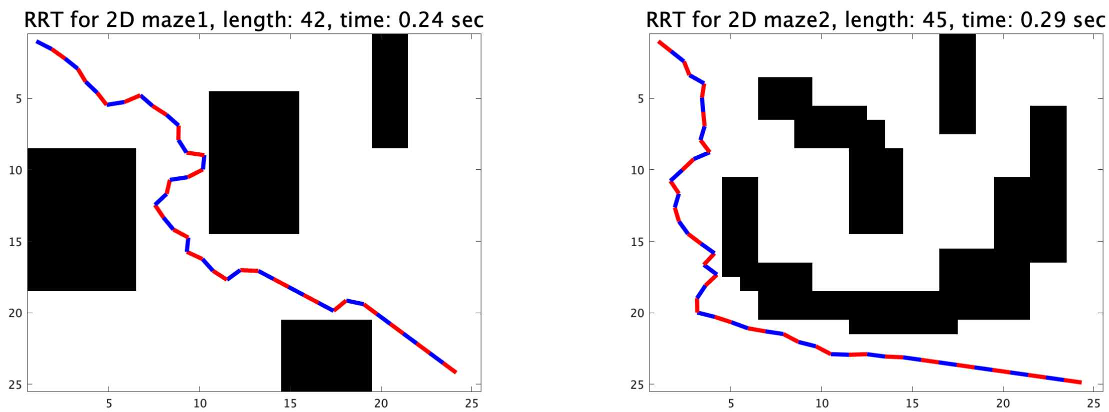


Figure 3: Planned path by greedy-A* for 2D maze environments. Note that euclidean distance is used as admissible heuristic. (a), (c), and (e) show planned paths for maze1 with time limits 0.05, 0.25, and 1 second. (b), (d), and (f) show planned paths for maze2 with time limits 0.05, 0.25, and 1 second.

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.



(a) (b)

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

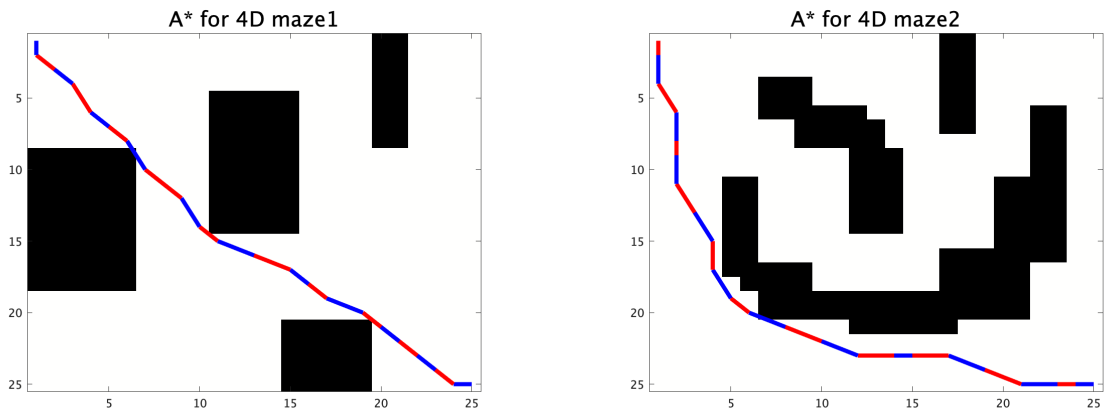
Table 5: User-specified parameters for RRT

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.



(a) (b)

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

ii

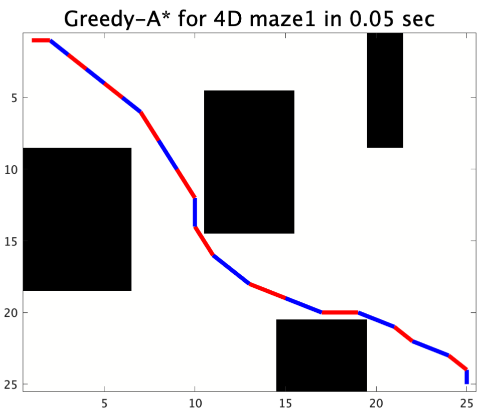
Table 6 and 7 summarize the number of nodes expanded and path length on maze 1 and maze 2 environments for each ϵ value, respectively. In addition, the paths found by greedy-A* implementation for 4D space are shown in Fig. 6. I used euclidean distance as admissible heuristic.

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

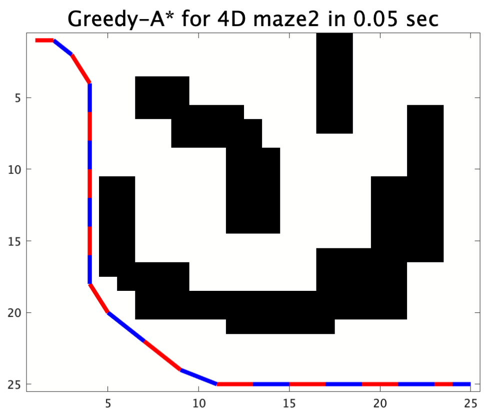
time limit: 0.05 sec			time limit: 0.25 sec			time limit: 1.0 sec		
ϵ	length	nodes	ϵ	length	nodes	ϵ	length	nodes
10	36.6	30	10	36.6	30	10	36.6	30
5.5	36.6	30	5.5	36.6	30	5.5	36.6	30
3.25	36.6	29	3.25	36.6	29	3.25	36.6	29
2.12	36.6	29	2.12	36.6	29	2.12	36.6	29
1.56	36.6	29	1.56	36.6	29	1.56	36.6	29
1.28	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.

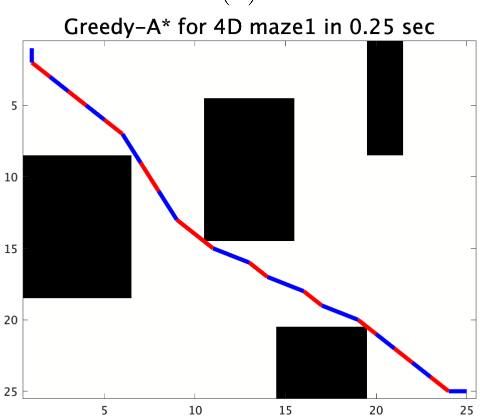
time limit: 0.05 sec			time limit: 0.25 sec			time limit: 1.0 sec		
ϵ	length	nodes	ϵ	length	nodes	ϵ	length	nodes
10	42.8	523	10	42.8	523	10	42.8	523
-	-	-	5.5	42.4	523	5.5	42.4	523
-	-	-	3.25	42.4	523	3.25	42.4	523
-	-	-	2.12	41.8	537	2.12	41.8	537
-	-	-	-	-	-	1.56	41.8	1193
-	-	-	-	-	-	1.28	41.1	1452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



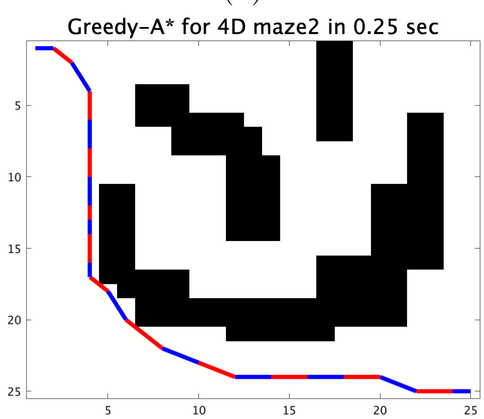
(a)



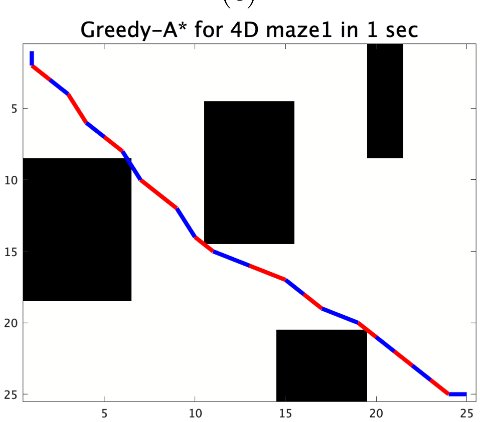
(b)



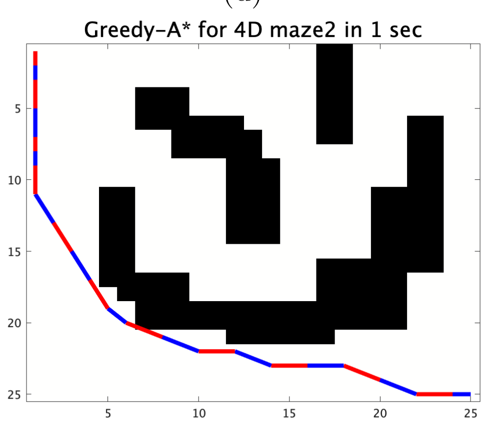
(c)



(d)



(e)



(f)

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

```
classdef Astar

    % set properties
    properties
        start
        goal
        map
        h_type
    end

    methods
        % constructor
        function obj = Astar(start, goal, map, h_type)
            obj.start = start;
            obj.goal = goal;
            obj.map = map;
            obj.h_type = h_type;
        end

        function [path] = search(obj)

            % initialization
            [~, num_nodes] = get_start(obj.map);
            priority_list = inf * ones(num_nodes, 1); % list containing priorities
            open_list = pq_init(1e+4); % open list
            closed_list = []; % closed list
            back_pointer = nan * ones(num_nodes, 1); % back-pointer attribute
            % starting position
            priority_list(obj.start) = h(obj, obj.start);
            open_list = pq_set(open_list, obj.start, priority_list(obj.start));
            back_pointer(obj.start) = -1;
            while true
                % pick best node and remove it from open_list
                [open_list, n_best] = pq_pop(open_list);
                % add best node to closed_list
                closed_list = [closed_list; n_best];

                % check n_best is goal or not
                if n_best == obj.goal
                    break
                end

                % expand all nodes that are neibhbors of n_best
                [neighbors, ~] = get_neighbors(obj.map, n_best);
                neighbors = neighbors((neighbors == n_best) ~= 1);
                for index_neighbor=1:length(neighbors)
                    neighbor = neighbors(index_neighbor);
                    % calculate each cost
                    cost_g = g(obj, n_best, priority_list);
                    cost_n = cost(obj, n_best, neighbor);
                    cost_h = h(obj, neighbor);
                    priority = cost_g + cost_n + cost_h;
                    % neighbor is in open_list
                    if pq_test(open_list, neighbor)
                        if priority < priority_list(neighbor)
                            % update open_list
                            priority_list(neighbor) = priority;
                            open_list = pq_set(open_list, neighbor, priority);
                            back_pointer(neighbor) = n_best;
                        end
                    % neighbor is not in open_list
                    else
```

```
                        if ismember(neighbor, closed_list)
                            continue
                        % add neighbor to open_list
                        else
                            priority_list(neighbor) = priority;
                            open_list = pq_set(open_list, neighbor, priority);
                            back_pointer(neighbor) = n_best;
                        end
                    end
                end
            end

            % after finishing search process, get an optimal path from solution
            path = get_path(obj, back_pointer);
        end

        function [path] = get_path(obj, back_pointer)
            % initialization
            [x_g, y_g] = state_from_index(obj.map, obj.goal);
            path = [x_g, y_g];
            n_best = back_pointer(obj.goal);
            while true
                [x_best, y_best] = state_from_index(obj.map, n_best);
                path = [path; x_best, y_best];
                n_best = back_pointer(n_best);
                if n_best == -1
                    break
                end
            end
        end

        % function to calculate actual cost of n_best
        function [cost_g] = g(obj, n_best, priority_list)
            cost_g = priority_list(n_best) - h(obj, n_best);
        end

        % function to calculate actual cost from n_best to neighbor
        function [cost_n] = cost(obj, n_best, neighbor)
            [x_best, y_best] = state_from_index(obj.map, n_best);
            [x_neig, y_neig] = state_from_index(obj.map, neighbor);
            cost_n = sqrt((x_neig - x_best)^2 + (y_neig - y_best)^2);
        end

        % function to calculate heuristic cost of the neighbor
        function [cost_h] = h(obj, neighbor)
            [x_neig, y_neig] = state_from_index(obj.map, neighbor);
            [x_goal, y_goal] = state_from_index(obj.map, get_goal(obj.map));
            if obj.h_type == 'm'
                % manhattan distance
                cost_h = abs(x_neig - x_goal) + abs(y_neig - y_goal);
            elseif obj.h_type == 'e'
                % euclidian distance
                cost_h = sqrt((x_neig - x_goal)^2 + (y_neig - y_goal)^2);
            else
                disp('select a type of heuristic function!')
            end
        end
    end
end
```

Greedy-A* Code for 2D Space

```
classdef greedy_Astar

    % set properties
```

```
    properties
        start
        goal
        map
        h_type
        epsilon
        t_limit
    end

    methods
        % constructor
        function obj = greedy_Astar(start, goal, map, h_type, epsilon, t_limit)
            obj.start = start;
            obj.goal = goal;
            obj.map = map;
            obj.h_type = h_type;
            obj.epsilon = epsilon;
            obj.t_limit = t_limit;
        end

        function [path] = greedy_search(obj)
            tic
            while toc < obj.t_limit
                [path, nodes_expanded] = search(obj);
                path_length = get_path_length(path);
                disp(' ')
                disp('-------------------')
                fprintf('epsilon: %f \n', obj.epsilon)
                fprintf('number of nodes expanded: %d, path length: %f \n', nodes_expanded,
                    path_length)
                disp('-------------------')
                disp(' ')

                obj.epsilon = obj.epsilon - 0.5 * (obj.epsilon - 1);
                if obj.epsilon == 1
                    break
                end
                if obj.epsilon < 1.001
                    obj.epsilon = 1;
                end
            end
        end

        function [path, nodes_expanded] = search(obj)

            % initialization
            [~, num_nodes] = get_start(obj.map);
            priority_list = inf * ones(num_nodes, 1); % list containing priorities
            open_list = pq_init(1e+4); % open list
            closed_list = []; % closed list
            back_pointer = nan * ones(num_nodes, 1); % back-pointer attribute
            % starting position
            priority_list(obj.start) = obj.epsilon * h(obj, obj.start);
            open_list = pq_set(open_list, obj.start, priority_list(obj.start));
            back_pointer(obj.start) = -1;
            while true
                % pick best node and remove it from open_list
                [open_list, n_best] = pq_pop(open_list);
                % add best node to closed_list
                closed_list = [closed_list; n_best];

                % check n_best is goal or not
                if n_best == obj.goal
                    break
                end

                % expand all nodes that are neibhbors of n_best
                [neighbors, ~] = get_neighbors(obj.map, n_best);
```

```
                neighbors = neighbors((neighbors == n_best) ~= 1);
                for index_neighbor=1:length(neighbors)
                    neighbor = neighbors(index_neighbor);
                    % calculate each cost
                    cost_g = g(obj, n_best, priority_list);
                    cost_n = cost(obj, n_best, neighbor);
                    cost_h = obj.epsilon * h(obj, neighbor);
                    priority = cost_g + cost_n + cost_h;
                    % neighbor is in open_list
                    if pq_test(open_list, neighbor)
                        if priority < priority_list(neighbor)
                            % update open_list
                            priority_list(neighbor) = priority;
                            open_list = pq_set(open_list, neighbor, priority);
                            back_pointer(neighbor) = n_best;
                        end
                    % neighbor is not in open_list
                    else
                        if ismember(neighbor, closed_list)
                            continue
                        % add neighbor to open_list
                        else
                            priority_list(neighbor) = priority;
                            open_list = pq_set(open_list, neighbor, priority);
                            back_pointer(neighbor) = n_best;
                        end
                    end
                end
            end

            % after finishing search process, get an optimal path from solution
            path = get_path(obj, back_pointer);
            nodes_expanded = length(closed_list);
        end

        function [path] = get_path(obj, back_pointer)
            % initialization
            [x_g, y_g] = state_from_index(obj.map, obj.goal);
            path = [x_g, y_g];
            n_best = back_pointer(obj.goal);
            while true
                [x_best, y_best] = state_from_index(obj.map, n_best);
                path = [path; x_best, y_best];
                n_best = back_pointer(n_best);
                if n_best == -1
                    break
                end
            end
        end

        % function to calculate actual cost of n_best
        function [cost_g] = g(obj, n_best, priority_list)
            cost_g = priority_list(n_best) - obj.epsilon * h(obj, n_best);
        end

        % function to calculate actual cost from n_best to neighbor
        function [cost_n] = cost(obj, n_best, neighbor)
            [x_best, y_best] = state_from_index(obj.map, n_best);
            [x_neig, y_neig] = state_from_index(obj.map, neighbor);
            cost_n = sqrt((x_neig - x_best)^2 + (y_neig - y_best)^2);
        end

        % function to calculate heuristic cost of the neighbor
        function [cost_h] = h(obj, neighbor)
            [x_neig, y_neig] = state_from_index(obj.map, neighbor);
            [x_goal, y_goal] = state_from_index(obj.map, get_goal(obj.map));
            if obj.h_type == 'm'
                % manhattan distance
```

```
                cost_h = abs(x_neig - x_goal) + abs(y_neig - y_goal);
            elseif obj.h_type == 'e'
                % euclidian distance
                cost_h = sqrt((x_neig - x_goal)^2 + (y_neig - y_goal)^2);
            else
                disp('select a type of heuristic function!')
            end
        end
    end
end
```

RRT Code for 2D Space

```
classdef RRT

    % set properties
    properties
        start
        goal
        map
        dist
        sampling_rate
        min_rand
        max_rand
    end

    methods
        % constructor
        function obj = RRT(start, goal, map, dist, sampling_rate, rand_range)
            obj.start = start;
            obj.goal = goal;
            obj.map = map;
            obj.dist = dist; % dist to steer sampled point
            obj.sampling_rate = sampling_rate; % rate to sample goal point
            obj.min_rand = rand_range(1);
            obj.max_rand = rand_range(2);
        end

        function [path, runtime] = search(obj)

            % initialization
            [x_start, y_start] = state_from_index(obj.map, obj.start);
            [x_goal, y_goal] = state_from_index(obj.map, obj.goal);
            node_list = [x_start, y_start];
            parent_list = [nan];
            node_cnt = 1;
            tic
            while true
                % random sampling
                [x_rand, y_rand] = get_node(obj);

                % find nearest node
                index_nearest = knnsearch(node_list, [x_rand, y_rand]);
                x_near = node_list(index_nearest, 1);
                y_near = node_list(index_nearest, 2);

                % expand tree towards [x_near, y_near]
                [x_new, y_new] = steer(obj, [x_near, y_near], [x_rand, y_rand]);

                % check collision
                dx = x_new - x_near;
                dy = y_new - y_near;
                if ~check_hit(obj.map, x_near, y_near, dx, dy)
                    % add new node
                    node_list = [node_list; x_new, y_new];
```

```
                    parent_list = [parent_list; index_nearest];
                    node_cnt = node_cnt + 1;
                end

                % check goal
                if abs(x_goal - x_new) < 1 && abs(y_goal - y_new) < 1
                    break
                elseif node_cnt >= 1e+4 % max iteration is 10000
                    disp('error: RRT could not reach goal within 10000 nodes')
                    break
                end
            end
            runtime = toc;
            % after finishing search process, get a path from solution
            path = get_path(obj, node_list, parent_list);
        end

        % function to get node
        function [x_rand, y_rand] = get_node(obj)
            if rand(1)> obj.sampling_rate
                x_rand = (obj.max_rand - obj.min_rand) * rand(1, 1) + obj.min_rand;
                y_rand = (obj.max_rand - obj.min_rand) * rand(1, 1) + obj.min_rand;
            else
                [x_goal, y_goal] = state_from_index(obj.map, obj.goal);
                x_rand = x_goal;
                y_rand = y_goal;
            end
        end

        % function to steer towards new node
        function [x_new, y_new] = steer(obj, node_near, node_rand)
            grad = atan2(node_rand(2) - node_near(2), node_rand(1) - node_near(1));
            dx = cos(grad) * obj.dist;
            dy = sin(grad) * obj.dist;
            x_new = node_near(1) + dx;
            y_new = node_near(2) + dy;
        end

        % function to get a final path from goal to start
        function [path] = get_path(obj, node_list, parent_list)

            % initialization
            path = [];
            index_node = length(node_list);
            while true
                path = [path; node_list(index_node, :)];
                index_node = parent_list(index_node, :);
                if isnan(index_node)
                    break
                end
            end
        end
    end
end
```

A* Code for 4D Space

```
classdef Astar_dynamic

    % set properties
    properties
        start
        goal
        map
        h_type
```

```
    end

    methods
        % constructor
        function obj = Astar_dynamic(start, goal, map, h_type)
            obj.start = start;
            obj.goal = goal;
            obj.map = map;
            obj.h_type = h_type;
        end

        function [path] = search(obj)

            % initialization
            [~, num_nodes] = get_start_dynamic(obj.map);
            priority_list = inf * ones(num_nodes, 1); % list containing priorities
            open_list = pq_init(1e+4); % open list
            closed_list = []; % closed list
            back_pointer = nan * ones(num_nodes, 1); % back-pointer attribute
            % starting position
            priority_list(obj.start) = h(obj, obj.start);
            open_list = pq_set(open_list, obj.start, priority_list(obj.start));
            back_pointer(obj.start) = -1;
            while true
                % pick best node and remove it from open_list
                [open_list, n_best] = pq_pop(open_list);
                % add best node to closed_list
                closed_list = [closed_list; n_best];

                % check n_best is goal or not
                if n_best == obj.goal
                    break
                end

                % expand all nodes that are neibhbors of n_best
                [neighbors, ~] = get_neighbors_dynamic(obj.map, n_best);
                neighbors = neighbors((neighbors == n_best) ~= 1);
                for index_neighbor=1:length(neighbors)
                    neighbor = neighbors(index_neighbor);
                    % calculate each cost
                    cost_g = g(obj, n_best, priority_list);
                    cost_n = cost(obj, n_best, neighbor);
                    cost_h = h(obj, neighbor);
                    priority = cost_g + cost_n + cost_h;
                    % neighbor is in open_list
                    if pq_test(open_list, neighbor)
                        if priority < priority_list(neighbor)
                            % update open_list
                            priority_list(neighbor) = priority;
                            open_list = pq_set(open_list, neighbor, priority);
                            back_pointer(neighbor) = n_best;
                        end
                    % neighbor is not in open_list
                    else
                        if ismember(neighbor, closed_list)
                            continue
                        % add neighbor to open_list
                        else
                            priority_list(neighbor) = priority;
                            open_list = pq_set(open_list, neighbor, priority);
                            back_pointer(neighbor) = n_best;
                        end
                    end
                end
            end

            % after finishing search process, get an optimal path from solution
            path = get_path(obj, back_pointer);
```

```
        end

        function [path] = get_path(obj, back_pointer)
            % initialization
            [x_g, y_g, ~, ~] = dynamic_state_from_index(obj.map, obj.goal);
            path = [x_g, y_g];
            n_best = back_pointer(obj.goal);
            while true
                [x_best, y_best, ~, ~] = dynamic_state_from_index(obj.map, n_best);
                path = [path; x_best, y_best];
                n_best = back_pointer(n_best);
                if n_best == -1
                    break
                end
            end
        end

        % function to calculate actual cost of n_best
        function [cost_g] = g(obj, n_best, priority_list)
            cost_g = priority_list(n_best) - h(obj, n_best);
        end

        % function to calculate actual cost from n_best to neighbor
        function [cost_n] = cost(obj, n_best, neighbor)
            [x_best, y_best, ~, ~] = dynamic_state_from_index(obj.map, n_best);
            [x_neig, y_neig, ~, ~] = dynamic_state_from_index(obj.map, neighbor);
            cost_n = sqrt((x_neig - x_best)^2 + (y_neig - y_best)^2);
        end

        % function to calculate heuristic cost of the neighbor
        function [cost_h] = h(obj, neighbor)
            [x_neig, y_neig, ~, ~] = dynamic_state_from_index(obj.map, neighbor);
            [x_goal, y_goal, ~, ~] = dynamic_state_from_index(obj.map, get_goal(obj.map));
            if obj.h_type == 'm'
                % manhattan distance
                cost_h = abs(x_neig - x_goal) + abs(y_neig - y_goal);
            elseif obj.h_type == 'e'
                % euclidian distance
                cost_h = sqrt((x_neig - x_goal)^2 + (y_neig - y_goal)^2);
            else
                disp('select a type of heuristic function!')
            end
        end
    end
end
```

Greedy-A* Code for 4D Space

```
classdef greedy_Astar_dynamic

    % set properties
    properties
        start
        goal
        map
        h_type
        epsilon
        t_limit
    end

    methods
        % constructor
        function obj = greedy_Astar_dynamic(start, goal, map, h_type, epsilon, t_limit)
            obj.start = start;
            obj.goal = goal;
```

```
            obj.map = map;
            obj.h_type = h_type;
            obj.epsilon = epsilon;
            obj.t_limit = t_limit;
        end

        function [path] = greedy_search(obj)
            tic
            while toc < obj.t_limit
                [path, nodes_expanded] = search(obj);
                path_length = get_path_length(path);
                disp(' ')
                disp('-------------------')
                fprintf('epsilon: %f \n', obj.epsilon)
                fprintf('number of nodes expanded: %d, path length: %f \n', nodes_expanded,
                    path_length)
                disp('-------------------')
                disp(' ')

                obj.epsilon = obj.epsilon - 0.5 * (obj.epsilon - 1);
                if obj.epsilon == 1
                    break
                end
                if obj.epsilon < 1.001
                    obj.epsilon = 1;
                end
            end
        end

        function [path, nodes_expanded] = search(obj)

            % initialization
            [~, num_nodes] = get_start_dynamic(obj.map);
            priority_list = inf * ones(num_nodes, 1); % list containing priorities
            open_list = pq_init(1e+4); % open list
            closed_list = []; % closed list
            back_pointer = nan * ones(num_nodes, 1); % back-pointer attribute
            % starting position
            priority_list(obj.start) = obj.epsilon * h(obj, obj.start);
            open_list = pq_set(open_list, obj.start, priority_list(obj.start));
            back_pointer(obj.start) = -1;
            while true
                % pick best node and remove it from open_list
                [open_list, n_best] = pq_pop(open_list);
                % add best node to closed_list
                closed_list = [closed_list; n_best];

                % check n_best is goal or not
                if n_best == obj.goal
                    break
                end

                % expand all nodes that are neibhbors of n_best
                [neighbors, ~] = get_neighbors_dynamic(obj.map, n_best);
                neighbors = neighbors((neighbors == n_best) ~= 1);
                for index_neighbor=1:length(neighbors)
                    neighbor = neighbors(index_neighbor);
                    % calculate each cost
                    cost_g = g(obj, n_best, priority_list);
                    cost_n = cost(obj, n_best, neighbor);
                    cost_h = obj.epsilon * h(obj, neighbor);
                    priority = cost_g + cost_n + cost_h;
                    % neighbor is in open_list
                    if pq_test(open_list, neighbor)
                        if priority < priority_list(neighbor)
                            % update open_list
                            priority_list(neighbor) = priority;
                            open_list = pq_set(open_list, neighbor, priority);
```

```
                            back_pointer(neighbor) = n_best;
                        end
                    % neighbor is not in open_list
                    else
                        if ismember(neighbor, closed_list)
                            continue
                        % add neighbor to open_list
                        else
                            priority_list(neighbor) = priority;
                            open_list = pq_set(open_list, neighbor, priority);
                            back_pointer(neighbor) = n_best;
                        end
                    end
                end
            end

            % after finishing search process, get an optimal path from solution
            path = get_path(obj, back_pointer);
            nodes_expanded = length(closed_list);
        end

        function [path] = get_path(obj, back_pointer)
            % initialization
            [x_g, y_g, ~, ~] = dynamic_state_from_index(obj.map, obj.goal);
            path = [x_g, y_g];
            n_best = back_pointer(obj.goal);
            while true
                [x_best, y_best, ~, ~] = dynamic_state_from_index(obj.map, n_best);
                path = [path; x_best, y_best];
                n_best = back_pointer(n_best);
                if n_best == -1
                    break
                end
            end
        end

        % function to calculate actual cost of n_best
        function [cost_g] = g(obj, n_best, priority_list)
            cost_g = priority_list(n_best) - obj.epsilon * h(obj, n_best);
        end

        % function to calculate actual cost from n_best to neighbor
        function [cost_n] = cost(obj, n_best, neighbor)
            [x_best, y_best, ~, ~] = dynamic_state_from_index(obj.map, n_best);
            [x_neig, y_neig, ~, ~] = dynamic_state_from_index(obj.map, neighbor);
            cost_n = sqrt((x_neig - x_best)^2 + (y_neig - y_best)^2);
        end

        % function to calculate heuristic cost of the neighbor
        function [cost_h] = h(obj, neighbor)
            [x_neig, y_neig, ~, ~] = dynamic_state_from_index(obj.map, neighbor);
            [x_goal, y_goal, ~, ~] = dynamic_state_from_index(obj.map, get_goal(obj.map));
            if obj.h_type == 'm'
                % manhattan distance
                cost_h = abs(x_neig - x_goal) + abs(y_neig - y_goal);
            elseif obj.h_type == 'e'
                % euclidian distance
                cost_h = sqrt((x_neig - x_goal)^2 + (y_neig - y_goal)^2);
            else
                disp('select a type of heuristic function!')
            end
        end
    end
end
```