Motion Planning

Ryosuke Kato

I. Introduction

Given global information, motion planning is a problem to determine the optimal path of the agent from initial point to the destination. The algorithms to solve the problem fall in mainly two categories: search-based motion planning and sampling-based motion planning. Many search-based planning algorithms are extension of classic deterministic shortest path (DSP) algorithms such as Dijkstra and A*. On the other hand, sampling-based planning is widely used in practice due to its time and memory efficiency.

II. Problem Statement

A. Deterministic Shortest Path Problem

Given continuous 3D space $\mathbb{R}^3$, we will solve deterministic shortest path (DSP) problem i.e. we will find the shortest path

$$i_{s:\tau} := (i_s, i_1, i_2, ..., i_\tau)$$

where i_s is the start node and i_τ is the destination node. Ideally, the detected path should minimizes the cost function $\mathcal{J}$,

$$\begin{aligned} i^*_{s:\tau} &= \text{argmin } \mathcal{J} \\ &= \text{argmin} \sum_{i=s}^{\tau-1} \text{cost}_{i,i+1} \end{aligned}$$

where $\text{cost}_{i,i+1}$ is cost from node i to j. The optimal cost achieved by the optimal path $i^*_{s:\tau}$ can be written as

$$\begin{aligned} dist(s,\tau) &= \min\ \mathcal{J} \\ &= \min \sum_{i=s}^{\tau-1} \text{cost}_{i,i+1} \end{aligned}$$

For each map, configuration space C consists of space occupied by obstacles $\mathcal{C}_{obs}$ and free space $\mathcal{C}_{free} = \mathcal{C} \backslash \mathcal{C}_{obs}$ and all nodes of the detected path should be inside free space i.e. $i \in \mathcal{C}_{free}$.

B. Constructing Graph

Before applying DSP finding algorithms, we need to construct a graph on which the algorithms traverse. We constructed three types of graph, (i) a 26-connected grid (by cell decomposition), (ii) a visibility graph (by skeltonization), and (iii) a continuous free space $\mathcal{C}_{free}$ as sampling space. Description of each graph and procedures to construct it will be discussed in **Technical Approach** section.

III. Technical Approach

In addition to collision checker, we implemented 4 algorithms (Combination of constructing a graph and search algorithm) to find the optimal path.

- Grid + Dijkstra (baseline)
- Grid + Weighted A*
- Visibility Graph + Dijkstra
- Rapidly Exploring Randomized Tree

A. Collision Checker

We implemented a collision checker of the given line segment with boxes from scratch. Since (i) all obstacles are rectangles and (ii) all faces of the boxes are parallel to x, y, or z axis, it is easier to implement it than more general settings.

The details of our collision checker is as follows. A box has 6 faces and each face can be represented as one of these three types,

- $\{(x,y,z) \mid x = c, y \in [y_{min}, y_{max}], z \in [z_{min}, z_{max}]\}$
- $\{(x,y,z) \mid y = c, x \in [x_{min}, x_{max}], z \in [z_{min}, z_{max}]\}$
- $\{(x,y,z) \mid z = c, y \in [x_{min}, x_{max}], y \in [y_{min}, y_{max}]\}$

where $c \in \mathbb{R}$ is some constant. Namely, exact one of three coordinate of a points on these planes is fixed. For example, suppose we are trying to check collision between line-segment whose endpoints are $\{(x_1,y_1,z_1),(x_2,y_2,z_2)\}$ with a plane $\{(x,y,z) \mid x = c, y \in [y_{min}, y_{max}], z \in [z_{min}, z_{max}]\}$. Then, the line-segment intersects with the plane if and only if all of the following three conditions hold

- $\min(x_1,x_2) \le c \le \max(x_1,x_2)$ (1)
- $y_{min} \le y^* \le y_{max}$ (if (1) holds) (2)
- $z_{min} \le z^* \le z_{max}$ (if (1) holds) (3)

where an intersection point of the line-segment and the plane is (c, y^*, z^*). Thus, the algorithm is as follows

Algorithm 1 Collision Checker

```
Input: a line segment, set of boxes
for each boxes do
    for each face in a box do
        if condition (1) holds then
            determine other two coordinates (y* and z* in the previous example)
            if condition (2) and (3) hold then return TRUE
return FALSE
```

Time complexity of the collision checker is $O(N)$ where N is number of obstacles.

B. Grid + Dijkstra

First we created a 26-connected grid graph and implemented Dijkstra algorithm run on it as a baseline model.

1) Constructing 26-connected Grid: For each map, we discretized a whole configuration space into $X \times Y \times Z$ voxels such that $X, Y, Z \in \mathbb{Z}$ voxels based on a resolution parameter λ. λ represents length of edges of voxels. For example, $\lambda = 0.1$ means length of edges of a voxel is 0.1 and $1 \times 1 \times 1$ unit cube is divided into $10 \times 10 \times 10 = 1000$ voxels in this case. Thus, the number of voxels is $O((\lambda^{-1})^3)$. We set $\lambda = 0.2$ for all grids through this project.

2) Dijkstra: We implemented Dijkstra algorithm working on this grid. Given current voxel (i, j, k) where $i, j, k \in \mathbb{Z}^+$ are indices of the voxel at 3D discrete coordinate, this algorithm searches 26 neighbor voxels $(i+a, j+b, k+c)$ i.e. $(a, b, c) \in \{a, b, c \mid a, b, c \in \{-1, 0, 1\}\}\backslash\{(0, 0, 0)\}$ at each step. The distance from a voxel (i, j, k) and a neighbor voxel $(i+a, j+b, k+c)$ is defined by L2 norm i.e.

$$distance((i, j, k), (i + a, j + b, k + c)) = \lambda\sqrt{a^2 + b^2 + c^2}$$

A state transition from (i_1, j_1, k_1) to (i_2, j_2, k_2) occurs only if (i) (i_2, j_2, k_2) is inside the boundary and (ii) a line-segment $\{(i_1, j_1, k_1), (i_2, j_2, k_2)\}$ does not intersect with any obstacle of the given map. We used collision checker which is previously mentioned here.

3) Property of Dijkstra: Dijkstra is a complete planning algorithm i.e. it searches all nodes $i \in \mathcal{C}_{free}$ until the optimal path is found and return FAIL if it is not found. Also, Dijkstra is optimal i.e. given a graph, Dijkstra always finds the shortest path if it exists. However, it is not efficient since it tries to search nodes in the opposite direction to τ even in homogeneous grid without obstacles. Due to such inefficiency, Dijkstra is not suitable to search high-dimensional space where number of nodes is exponentially huge. Its time complexity is $O(|\mathcal{V}| \log |\mathcal{V}|) = O(N^3 \log N^3)$, N: number of nodes in each dimension (i) on sparse graph such as grid and (ii) using Binary heap. Since it requires information such as distance from the start node and parent node for each node, its memory complexity is $O(N^3)$.

C. Grid + Weighted A*

1) Weighted A:* Next, we implemented weighted A* algorithm working on the same discretized 26-connected grid. Weighted A* is generalized version of A* algorithm and it is equivalent to A* if weight parameter $\epsilon = 1$. A* and weighted A* is more efficient than Dijkstra by guidance of heuristic function h. To implement it, we only need a few modification to change Dijkstra algorithm to weighted A* algorithm. The difference of these algorithms are as follows. In Dijkstra, the condition to update distance of node u is

$$distance(s, u) > distance(s, v) + c_{v,u}$$

Meanwhile, in weighted A*, the condition to update distance of node u is

$$distance(s, u) > distance(s, v) + c_{v,u} + \epsilon h(u)$$

where $v = parent(u)$ and $distance(s, u)$ denotes distance from start node to node u. $c_{v,u}$ is distance from v to u. $\epsilon \geq 1$ is weight parameter to determine how important we consider the output value of heuristic function of node u, $h(u)$. We use various value for parameter ϵ and use L2 norm from current node i to the goal node τ, $\|x_i - x_\tau\|_2$ as heuristic function. In addition, weighted A* algorithm pushes $distance(s, u) + \epsilon h(u)$ as a key of node u into priority queue $OPEN$ while Dijkstra uses $distance(s, u)$ as a key in the queue.

2) Property of weighted A:* Normal A* ($\epsilon = 1$) is complete. It is also optimal under condition that heuristic function is admissible. L2 norm heuristic function we used in this project is admissible in a 26-connected grid graph. However, weighted $A*(\epsilon > 1)$ does not satisfy these properties because L2 norm heuristic function is no more admissible if $\epsilon > 1$. As ϵ increases, number of nodes which is not considered increases and thus the found path tend to be farther from the optimal. In trade-off for these disadvantages, weighted A* is more efficient than normal A*. Normal A* is still much more efficient than Dijkstra. Though time and space complexity of A* and weighted A* are the same as those of Dijkstra in the worst case of i.e. $O(N^3 \log N^3)$ and $O(N^3)$ respectively, the actual complexity is usually much less than those.

D. Visibility Graph + Dijkstra

When we were running Dijkstra and weighted A* on the grid graph, we noticed that the shortest path consists of line segments whose end points are always one of start node, goal node, corner of boxes. Thus, we think it can be more efficient to construct a visibility graph and run some DSP algorithms such as Dijkstra on it. A visibility graph is defined as a graph whose vertices are located on one of start node, goal node, or corner of obstacles and whose edges are line segments between vertices without any obstacles on it. Its main advantage is that since it is defined on continuous space $\mathbb{R}^3$, its performance does not depend on number of voxels of grid and it is scalable.

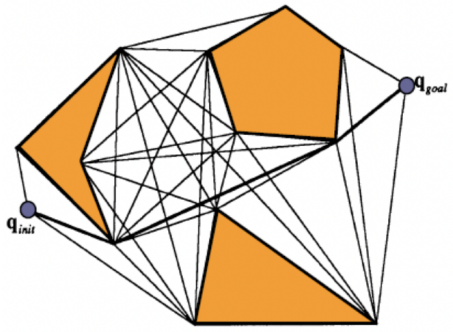


Fig. 1. Visibility Graph

1) Constructing visibility graph: To construct visibility graph, we introduced two additional parameters, λ_2 and λ_3. Since an agent can not go through exactly on a corner of obstacles, it should turn around the corner with some small margin distance to the corner and λ_2 represents this small distance. We fixed $\lambda_2 = 0.2$.

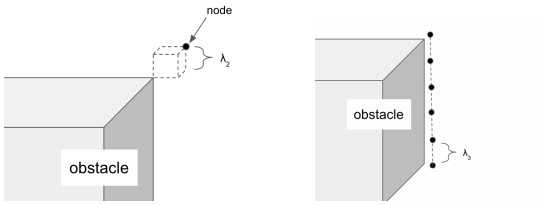


Fig. 2. Margin parameter λ_2 and edge division parameter λ_3

In addition, we need also create vertex of visibility graph on several point of each edge of obstacle and λ_3 represents how small vertices of visibility graph we divide an edge of obstacle into. For example, if $\lambda_3 = 0.2$, we create $5 + 1$ vertices between two corner $(0, 0, 0)$ and $(0, 0, 1)$ of a box i.e. created vertices are $\{(0, 0, 0), (0, 0, 0.2), (0, 0, 0.4), (0, 0, 0.6), (0, 0, 0.8), (0, 0, 1)\}$. Then, for all pairs of these created vertices, we check if a line segments connecting each pair is collision-free and if it is collision free, created an edge of visibility graph. We also used our collision checker here. The whole picture of creating visibility graph algorithm is as follows,

We used two value of λ_3, $\lambda_3 = 0.2$ and 0.5. Time complexity of constructing visibility graph is $O(N^3\lambda_3^{-1})$ (each pair of node: $O(N^2)$ and collision check: $O(N)$) where N is number of obstacles. Since $O(N^3\lambda_3^{-1})$ is expensive to some extent, we will see constructing visibility graph becomes bottle-neck especially in environment where there are many obstacles such as room or maze.

2) Dijkstra on visibility graph: Then, we used Dijkstra algorithm to find the shortest path on the visibility graph. Since number of nodes of visibility graph is at most $2,000$ as shown in table 2 and thus computing time of our Dijkstra is relatively shorter than time to create visibility graph. So, we only used Dijkstra and did not use another such as A* to seek for shorter time complexity.

Algorithm 2 Constructing Visibility Graph

Input: start node s, goal node τ, set of obstacles $\mathcal{O}$
Initialize set of vertices $\mathcal{V} = \{s, \tau\}$
for each obstacle **do**
  for each edge of the obstacle **do**
    Divide an edge to vertices $v_1, ..., v_n$ based on λ_3
    Add $v_1, ..., v_n$ to $\mathcal{V}$
Initialize set of edges $\mathcal{E}$: EMPTY
for each pair of nodes (v_i, v_j), s.t. $v_i, v_j \in \mathcal{V}$ **do**
  if line segment (v_i, v_j) is collision free **then**
    Create edge $e_{i,j}$ and add $e_{i,j}$ to $\mathcal{E}$
return Visibility graph $(\mathcal{V}, \mathcal{E})$

E. Rapidly Exploring Randomized Tree

We also used rapidly exploring randomized tree (RRT) as sampling-based planning algorithm. To implement it, we used python package `rrt-algorithms`. At each step, RRT samples a new configuration point x_{rand} and determine $x_{nearest}$, the nearest node to x_{rand} from the existing tree add a new node such that

$$x_{new} \leftarrow \text{STEER}_\epsilon(x_{nearest}, x_{rand}),$$

where ϵ is edge length parameter, to the tree as a new node or create a node on a line segment between the nearest $x_{nearest}$ and x_{rand} if $x_{rand} \in \mathcal{C}_{obs}$. RRT also checks if the existing tree from start node x_s is connected to x_τ with some small probability p_{check} at each step. We set $p_{check} = 0.1$, number of samples $= 2^{20}$, and changed the length of edges of tree $l = \{1, 2, 3\}$. RRT is probabilistically complete and in general, its time and space complexity is much less than those of Dijkstra or weighted A*. However, it is not optimal.

IV. Results

(i) The optimal cost and computing time, (ii) considered nodes and (iii) trajectory of optimal path for each map for each algorithm are shown in Table 1, Table 2, and Figure 3 (in the last page of this report), repsectively. Overall, Visibility Graph + Dijkstra shows the best performance among them and Grid + weighted A* follows it.

A. Grid + Dijkstra

Grid + Dijkstra finds the optimal paths at each map. However, it spends more time than other algorithm and spends 20 seconds even in a simple map with one obstacle (single cube). As shown in table 2, the number of nodes considered by Dijkstra is much more than (weighted) A* in the same grid graph and it is main reason of efficiency.

B. Grid + (Weighted) A*

Normal A* ($\epsilon = 1$) achieves the optimal path as well as shorter computing time and therefore, it can be thought as one of the most preferable algorithms among ones used in this project. It reduces time significantly especially in a simple map such as single cube. Table 2 on considered nodes also shown how efficiently (weighted) A* checks nodes comparing to plain Dijkstra. In contrast, in more complicated maps such as Monza or Maze in which there are many obstacles and the optimal path is zig-zag, its computing time is slightly less than that of Dijkstra. If $\epsilon > 1$, the path is not the optimal but it is still near-optimal in shorter computing time. Therefore, weighted $A*(\epsilon > 1)$ can be suitable in the case that we give priority time over cost.

C. Visibility Graph + Dijkstra

Visibility Graph + Dijkstra is the most successful algorithm in this project. Since a visibility graph is, unlike a grid graph, constructed in continuous space $\mathbb{R}^3$, two node of the path, which are located diagonally each other, can be connected directly and the cost of the optimal path can be better than Grid + Dijkstra. As shown in figure 3, since it can connect two nodes far from each other directly, it is more efficient than algorithms on a grid graph especially in simple settings such as a single cube. Main disadvantage of Visibility Graph + Dijkstra is that it time complexity is much worse than other algorithms on a grid graph in environment where many object exist such as room or window. In these complex environment, constructing visibility graph becomes a burden due to its time complexity $O(\lambda_3^{-1}N^3)$. As resolution parameter λ_3 becomes large, the computing time significantly decreases while the solution is slightly worse. Especially, it finds a path in Maze in 13 seconds.

D. Rapidly Randomized Tree

RRT finds paths much faster than other algorithms especially when there are many obstacles while there can be several paths such as room or maze. Its main disadvantage is environment which there can be only one or a few paths and it goes through narrow passages. For example, in Monza, it needs to consider 3,000 - 10,000 nodes to find a path which goes through three small passages. Likewise to visibility graph, it can treat continuous space without discretization, it can find a path to far destination in short computing time. The detected path is not optimal. As shown in figure 3, its path looks like random walk; it move forward and backward even if there is no obstacles. Such random walk makes the cost of path detected by RRT much worse than other algorithms. As edge length parameter l increases, the computing time is improved in a complex map (Monza) while the optimal cost tends to be worse, in general.

Algorithm		SC	R	MO	FB	T	W	MA
G + D	Cost	11.69	11.47	76.26	**25.43**	**28.49**	**23.51**	**86.16**
	Time	20.35	10.35	4.95	8.14	12.87	21.10	102.36
G + WA	Cost	11.69	11.82	76.42	**25.43**	**28.49**	**23.51**	**86.82**
($\epsilon = 1$)	Time	0.20	2.13	4.15	5.46	9.10	1.72	63.71
G + WA	Cost	12.01	12.50	77.39	29.74	34.76	24.40	87.85
($\epsilon = 1.5$)	Time	0.15	0.62	3.89	3.31	7.71	**0.10**	47.26
G + WA	Cost	12.01	11.93	77.35	28.99	35.44	24.40	88.04
($\epsilon = 2$)	Time	0.12	**0.52**	3.93	1.15	**2.11**	**0.15**	46.41
VG + D	Cost	**11.40**	**10.97**	**74.88**	26.32	**29.31**	**23.16**	**85.94**
($\lambda_3 = 0.2$)	Time	**0.024**	36.87	**0.17**	1.18	16.86	14.84	104.42
VG + D	Cost	**11.48**	11.61	78.45	30.30	32.82	**23.73**	93.24
($\lambda_3 = 0.5$)	Time	**0.025**	6.47	**0.034**	**0.24**	**2.86**	3.20	**13.84**
RRT	Cost	12.84	23.37	106.18	41.90	45.74	34.75	150.80
($l = 1$)	Time	0.056	**0.47**	73.10	1.02	**2.49**	1.19	26.57
RRT	Cost	13.05	21.85	106.03	52.95	54.22	36.32	166.86
($l = 2$)	Time	0.067	**0.35**	54.53	1.26	3.12	1.27	42.28
RRT	Cost	13.18	26.31	124.49	51.29	50.19	41.43	155.67
($l = 3$)	Time	0.060	0.76	50.07	1.77	3.13	1.62	34.86

TABLE I
OPTIMAL COST AND COMPUTING TIME

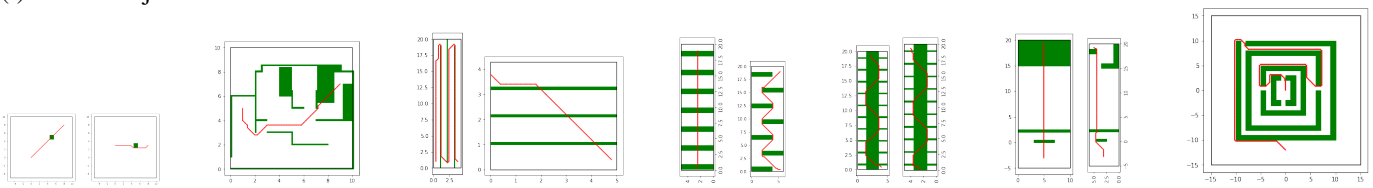
SC: Single Cube, R: Room, MO: Monza, FB: Flappy Bird, T: Tower, W: Window, MA: Maze, G + D: Grid + Dijkstra, G + WA: Grid + Weighted A*, VG + D: Visibility Graph + Dijkstra, RRT: Rapidly Randomized Tree

Algorithm	SC	R	MO	FB	T	W	MA
G + D	367,828	25,158	50,068	61,567	44,705	171,377	448,310
G + WA ($\epsilon = 1$)	3,104	7,046	39,536	45,141	35,287	18,604	249,450
G + WA ($\epsilon = 1.5$)	651	2,283	37,114	31,270	31,264	1,314	157,591
G + WA ($\epsilon = 2$)	621	2,073	37,086	11,087	8,236	1,259	153,250
VG + D ($\lambda_3 = 0.2$)	66	711	152	327	951	810	1,952
VG + D ($\lambda_3 = 0.5$)	46	290	68	145	376	384	717
RRT ($l = 1$)	9	90	17,079	293	546	298	5,975
RRT ($l = 2$)	7	28	5,693	191	277	160	3,360
RRT ($l = 3$)	3	41	3,389	172	164	169	1,792

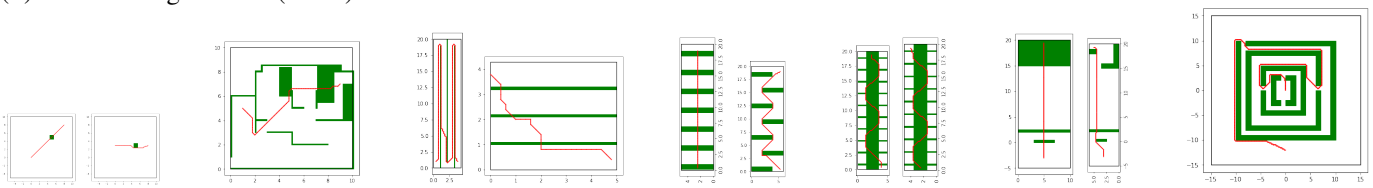
TABLE II
NUMBER OF CONSIDERED NODES

Fig. 3. (Next page) Computed paths by each algorithm in 2D map projected from 3D map
(From left to right) (a) Single Cube $\times$ 2, (b) Room, (c) Monza $\times$ 2, (d) Fluppy Bird $\times$ 2, (e) Tower $\times$ 2, (f) Window $\times$ 2, (g) Maze

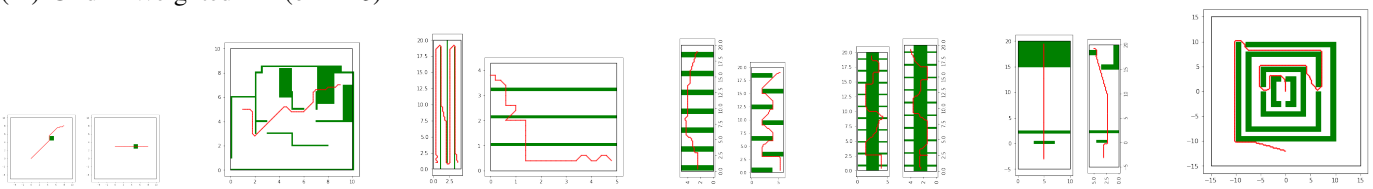
(i) Grid + Dijkstra



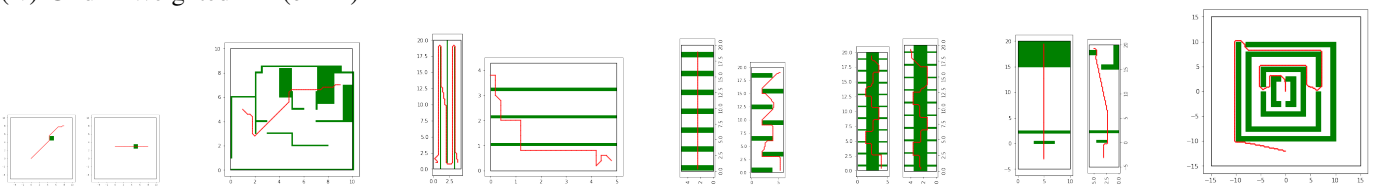
(ii) Grid + Weighted A* ($\epsilon = 1$)



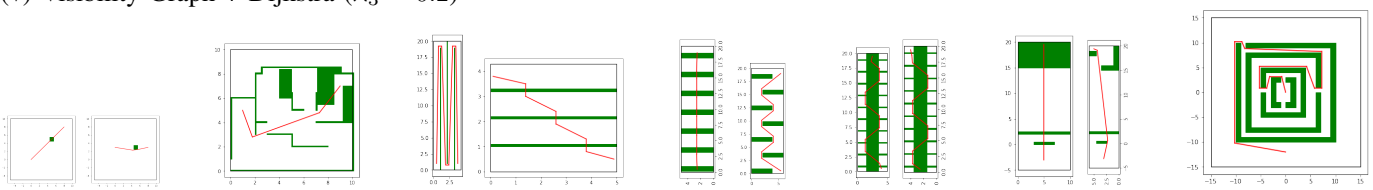
(iii) Grid + Weighted A* ($\epsilon = 1.5$)



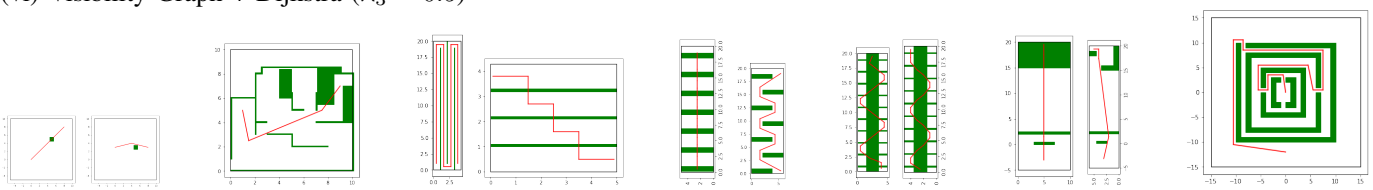
(iv) Grid + Weighted A* ($\epsilon = 2$)



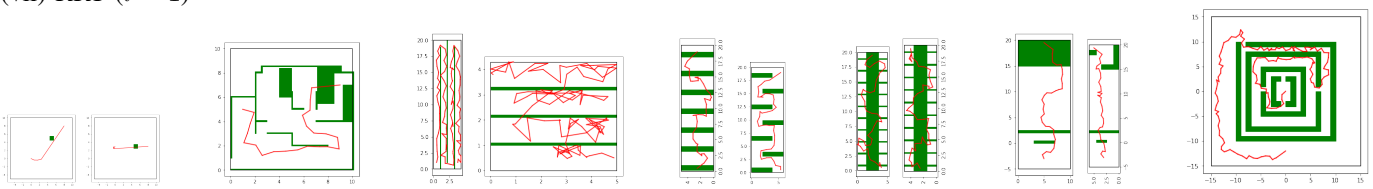
(v) Visibility Graph + Dijkstra ($\lambda_3 = 0.2$)



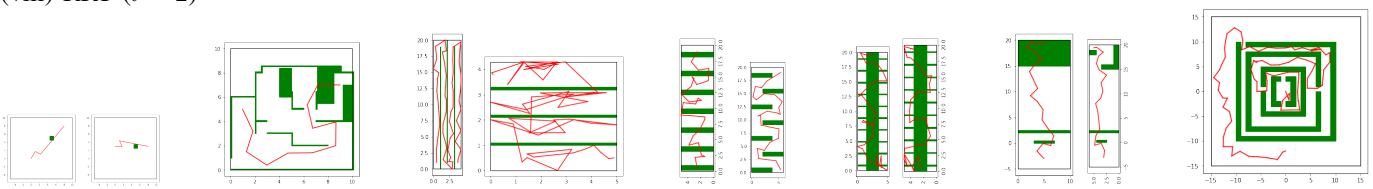
(vi) Visibility Graph + Dijkstra ($\lambda_3 = 0.5$)



(vii) RRT ($l = 1$)



(viii) RRT ($l = 2$)



(ix) RRT ($l = 3$)