Instruction for Assignment 2 for Term Project

Rapidly-exploring Random Tree and Path Planning

Introduction

The objective of this semester's term project is to implement a path planning algorithm for a mobile robot in a dynamic environment. This project is a simplified version of the IROS Kinect Robot Navigation Contest 2014. In that contest, participants use state-of-art mapping and planning algorithms. However, in this term project, the participants are ordered to implement two of the most commonly used path planning algorithms named Rapidly-exploring Random Tree (RRT) and its variation RRT*. For this week, we will focus on the RRT.

Rapidly-exploring Random Tree (RRT)

A Rapidly-exploring Random Tree (RRT) is a data structure and path planning algorithm that is designed for efficiently searching paths in nonconvex high-dimensional spaces. RRTs are constructed incrementally by expanding the tree to a randomly-sampled point in the configuration space while satisfying given constraints, e.g., incorporating obstacles or dynamic constraints (nonholonomic or kinodynamic constraints). While an RRT algorithm can effectively find a feasible path, an RRT alone may not be appropriate to solve a path planning problem for a mobile robot as it cannot incorporate additional cost information such as smoothness or length of the path. Thus, it can be considered as a component that can be incorporated into the development of a variety of different planning algorithms, e.g., RRT*.

Algorithm

A brief description of the RRT for a general configuration space is shown in Figure 1. An RRT rooted at a configuration x_{init} and has K-vertices is constructed using the following algorithm.

```
GENERATE_RRT($x_{init}, K, \Delta t$)
1  $\mathcal{T}$.init($x_{init}$);
2  for $k = 1$ to $K$ do
3      $x_{rand} \leftarrow$ RANDOM_STATE();
4      $x_{near} \leftarrow$ NEAREST_NEIGHBOR($x_{rand}, \mathcal{T}$);
5      $u \leftarrow$ SELECT_INPUT($x_{rand}, x_{near}$);
6      $x_{new} \leftarrow$ NEW_STATE($x_{near}, u, \Delta t$);
7      $\mathcal{T}$.add_vertex($x_{new}$);
8      $\mathcal{T}$.add_edge($x_{near}, x_{new}, u$);
9  Return $\mathcal{T}$
```

Figure 1. Pseudo code of an RRT

x_{init} indicates an initial position of a robot in the Cartesian coordinate. K indicates the number of vertices of a tree and the algorithm iterates K times before termination. This loop termination condition can be substituted by checking the closest distance from the tree to the goal point. To implement this, you should use 'while loop' instead of 'for loop'. Additionally you can make your loop iterate at least K times before termination. Δt indicates time interval and $\mathcal{T}$ represents a tree structure which contains nodes sampled from configuration space and u indicates the control input. C, C_{tree} and C_{obs} indicates configuration space, free configuration space and obstructed configuration space, respectively. In our project, C is a given as a 2D map and we will not consider control (input) state in the project. Instead, the pure-pursuit algorithm will be used to determine control (input) of the robot to follow the path given by the RRT algorithm. In the following pargraphs, we will explain each step of RRT algorithm.

Step 1. Initialize a tree to have its root as an initial position of a robot.

Step 3. Choose a random position x_{rand} in C (configuration space). Alternatively, one could replace RANDOM_STATE with RANDOM_FREE_STATE, and sample configuration in C_{tree} (by using a collision detection algorithm to reject samples from C_{obs}).

Step 4. Select the vertex, x_{near}, in the tree that is closest to x_{rand}.

Step 5 and 6. NEW_STATE selects a new configuration, x_{new}, that is away from x_{near} by an incremental distance, Δx, toward the direction of x_{rand}. The function SELECT_INPUT generates a control (input) that moves the robot from x_{near} toward x_{new}. However, in this project, we will ignore this step as the control will be generated using the pure-pursuit algorithm.

Step 7 and 8. Add x_{new} vertex and edge to the RRT.

Figure 2 illustrates the tree expansion mechanism.

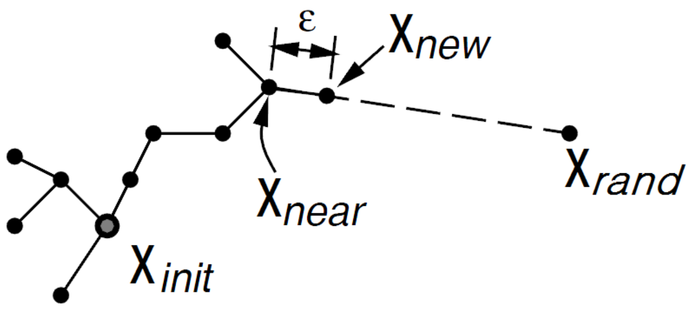

Figure 2. Mechanism of tree expansion of an RRT

For better understanding of RRTs, consider the special case where C is a square region in the plane. Let ρ represent the Euclidean metric. Figure 3 illustrates the construction of an RRT for the case of $C = [0, 100] \times [0, 100]$, $\Delta x = 1$, and $x_{init} = (50, 50)$:

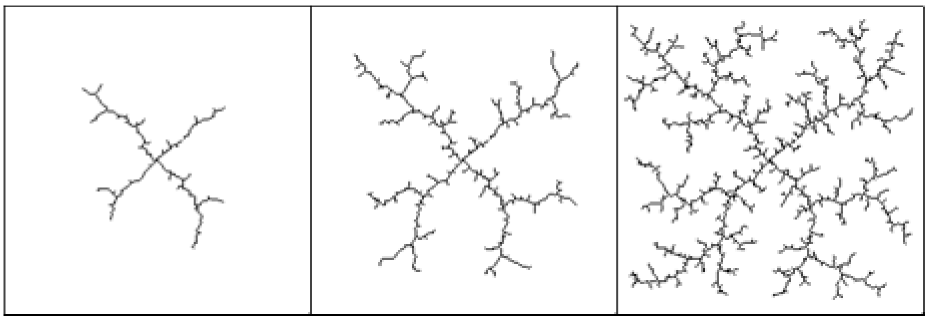

Figure 3. Example of construction of an RRT in a squared configuration space

The RRT quickly expands in a few directions to quickly explore the four corners of the square. Although the construction method is simple, it is no easy task to find a method that yields such desirable behavior. It is because an expansion of an RRT is being biased toward places not yet visited.

Modified Algorithm (Goal Bias)

In the actual implementation of the RRT algorithm, a simple modification named 'goal bias' is required. In selecting the random point, x_{rand}, in the configuration space in Step 3, a goal bias modification simply selects the goal point with pre-fixed frequency, e.g., once every five iterations. With this simple modification, we can make the tree to expand to the goal point. Note that this is closely related to the concept of exploration and exploitation widely used in a machine learning literature.

Skeleton Code

rrtTree.cpp

rrtTree(point x_init, point x_goal, cv::Mat map, double map_origin_x, double map_origin_y, double res, int margin)

: this function is constructor of rrtTree class. This constructor initialize member variable of rrtTree class. map_origin_x, map_origin_y means translation between global coordinate to grid map coordinate. And res means resolution of map(0.05m), margin means margin of obstacles. You don't need to implement this function, just use this in main.cpp.

void addVertex(point x_new, point x_rand, int idx_near)

: In this function, add new vertex to tree. Tree structure is array of node. The function of addVertex is expressed in Figure 4 and Figure 5. Figure 4 shows how to add new node D in tree.

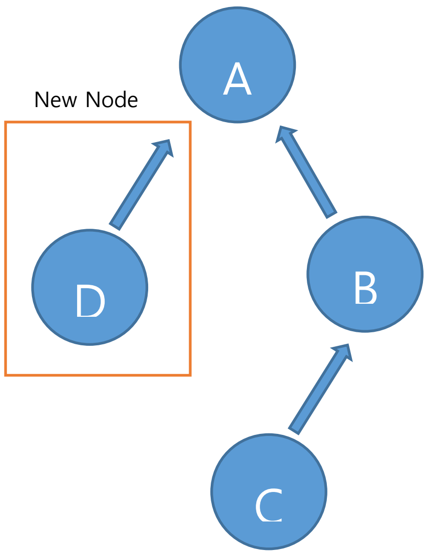

Figure 4 Add a new node

Figure 4 shows the situation adding node D to parent node A. Original tree has node A, node B and node C. and these nodes are in node array of tree class. Figure 5 shows array structure which actually store tree nodes. And parent_idx means the index of parent node in this array. When you add a new node like Figure 4, the parent_idx of a new node will be 0 like Figure 5.

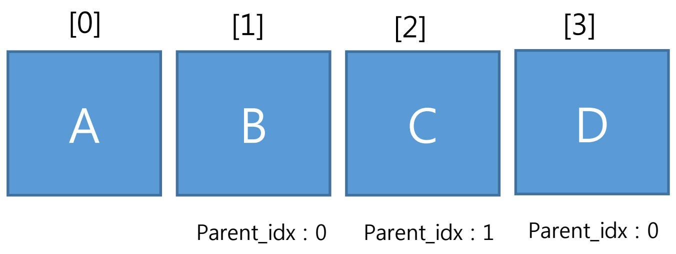

Figure 5 Array structure of tree

int nearestNeighbor(point x_rand)

bool isCollision(point x1, point x2)

: In this collision check function, you will need to use a class member variable cv::Mat map. You may regard the type of cv::Mat as a matrix. You can access to the (i, j) element by

```
map_margin.at<uchar>(i, j)
```

This expression will be used to read and write value of (i, j) element in cv::Mat map. The value of the grid map indicates the occupancy state where 0 indicates free and 255 indicates occupied. If the region is unknown, it has value 125. Each cell in a grid map represent a square region of 5cm by 5cm.

For convenience, it will be helpful to think the Cartesian coordinate as Figure 4. The row of the grid map coincides with x and column of it coincides with y. And the origin will be the center point of the occupancy grid map, i.e., (400.5, 400.5) for 800 x 800 grid.

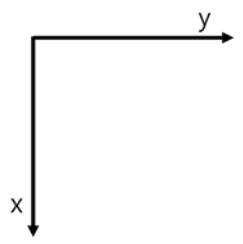

Figure 6. Axes for the project

i = x/res + origin_x

j = y/res + origin_y

point randomState(double x_max, double x_min, double y_max, double y_min)

: x_max, x_min, y_max, y_min indicates the size of a map with respect to real world coordinate. In this function, you need to randomly sample a point in real world. Sampling space is rectangle $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$

int generateRRT(double x_max, double x_min, double y_max, double y_min, int K, double MaxStep)

: x_max, x_min, y_max, y_min indicates the size of real world coordinate. You have to call this function in **TODO part of void generate_path_RRT()** of main.cpp. You can use following variables in main.cpp.

double world_x_min;

double world_x_max;

double world_y_min;

double world_y_max;

assign x_max, x_min, y_max, y_min as world_x_max, world_x_min, world_y_max, world_y_min.

K is minimum iteration number.

point newState(int idx_near, point x_rand, double MaxStep)

: MaxStep is maximum distance from x_near to x_new. idx_near is index of the nearest node in the node array from the x_rand. And you should make new point like Figure 2.

std::vector<point> backtracking()

: Those who are not familiar with a vector class, see
http://www.cplusplus.com/reference/vector/vector/?kw=vector
Return the vector containing path extracted from an RRT in a reverse order (goal to initial). Find the nearest leaf node from the goal and track parents of nodes iteratively.

void visualizeTree() and void visualizeTree(std::vector<point> path)

: For the debugging purpose, class member function void visualizeTree(), and
void visualizeTree(std::vector path) is provided. You can call this function inside the class. And if you want to highlight a specific path, use void visualizeTree(std::vector path).

main.cpp

In main.cpp, you should make a finite state machine. Flow of main.cpp is likely to following.

1. generate path.

2. tracking generated path using pure pursuit controller.

void generate_path_RRT()

You have to generate a path which connects all way point in sequence. Way points are stored in variable “waypoints” whose type is std::vector<point>. So you iteratively generate each path which connects consecutive two way points and gather them in variable “path_RRT”

RUNNING state

You should implement RUNNING state in finite state machine. And this machine has three state, “INIT”, “RUNNING” and “FINISH”. You just implement TODO part in RUNNING state. The other states are already implemented. In RUNNING state, make control tracking current pursuit point of generated path. And check distance between a current tracking point and robot. If distance is less than 0.2, then update current look_ahead_idx as next index (look_ahead_idx++). And check whether robot reach the goal or not. If robot reach the goal within 0.2 error. Stop finite state machine by changing state to FINISH.

Submission Format

Compress your project folder including all your project files and upload it on the eTL. The name of the compressed file should be **“IS_Project_02_[TeamName].tar.gz”**.

Reference

[1] LaValle, Steven M. "Rapidly-Exploring Random Trees A New Tool for Path Planning." (1998).