Instruction for Assignment 3 for Term Project

Implementation of RRT-Star and Sensing with Kinect for Dynamic Mapping

Introduction

If you have accomplished the previous assignments until now, you fully understood RRT. Although, the RRT performs well in a sense of finding a path to the goal, it does not guarantee the optimality of the path, as you can figure out there is no other constraint but an obstacle check. If we want to extract a path satisfying desirable condition, you can design some cost function that maps a path to some value and can use it as a measure of an optimality. Then the problem becomes an optimization problem resulting path which minimizes the cost function under the constraints, i.e., configuration space with obstacles. Taking the advantage of this concept, an RRT-Star can output an optimal path to the goal point. Now it's your turn to taste this manner.

RRT-Star

Algorithm 6: RRT*

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$;
    $V \leftarrow V \cup \{x_{\text{new}}\}$;
    $x_{\min} \leftarrow x_{\text{nearest}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Connect along a minimum-cost path
      if $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
        $x_{\min} \leftarrow x_{\text{near}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
    $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Rewire the tree
      if $\texttt{CollisionFree}(x_{\text{new}}, x_{\text{near}}) \wedge \texttt{Cost}(x_{\text{new}}) + c(\texttt{Line}(x_{\text{new}}, x_{\text{near}})) < \texttt{Cost}(x_{\text{near}})$ **then** $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
      $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
return $G = (V, E)$;

Figure 1. Pseudo code of RRT-Star

Figure 1 describes the procedure of an RRT-Star. As you can see in the line 7, **you need to determine parameters γ_{RRT} and η as well as the cardinality of V to decide the region of X_{near}. However, since the exact parameter setting guaranteeing the optimality is a tedious task, you can substitute this part with your own heuristic.** Although, this substitution may deteriorate the performance of guaranteeing an optimality of the output path, this trade-off is a widely used manner that an engineer would have in mind. Think about which heuristic you can use for the easy implementation.

Second, we throw a question about when a new node, depicted in dark blue in figure 2, is added to a tree. The longest edge in the top is an updated edge after line 14-16 is executed in the previous iteration of line 2. Suppose we don't have intermediate node depicted in dashed circle, then which node will become a parent node of X_{new}? And as you can inspire from the figure 2, which one is the desirable shortest path among three dashed arrows? Sometimes, heuristics contribute a lot making your algorithm result better performance. Think about which heuristic can be applied to your own troublesome problems.

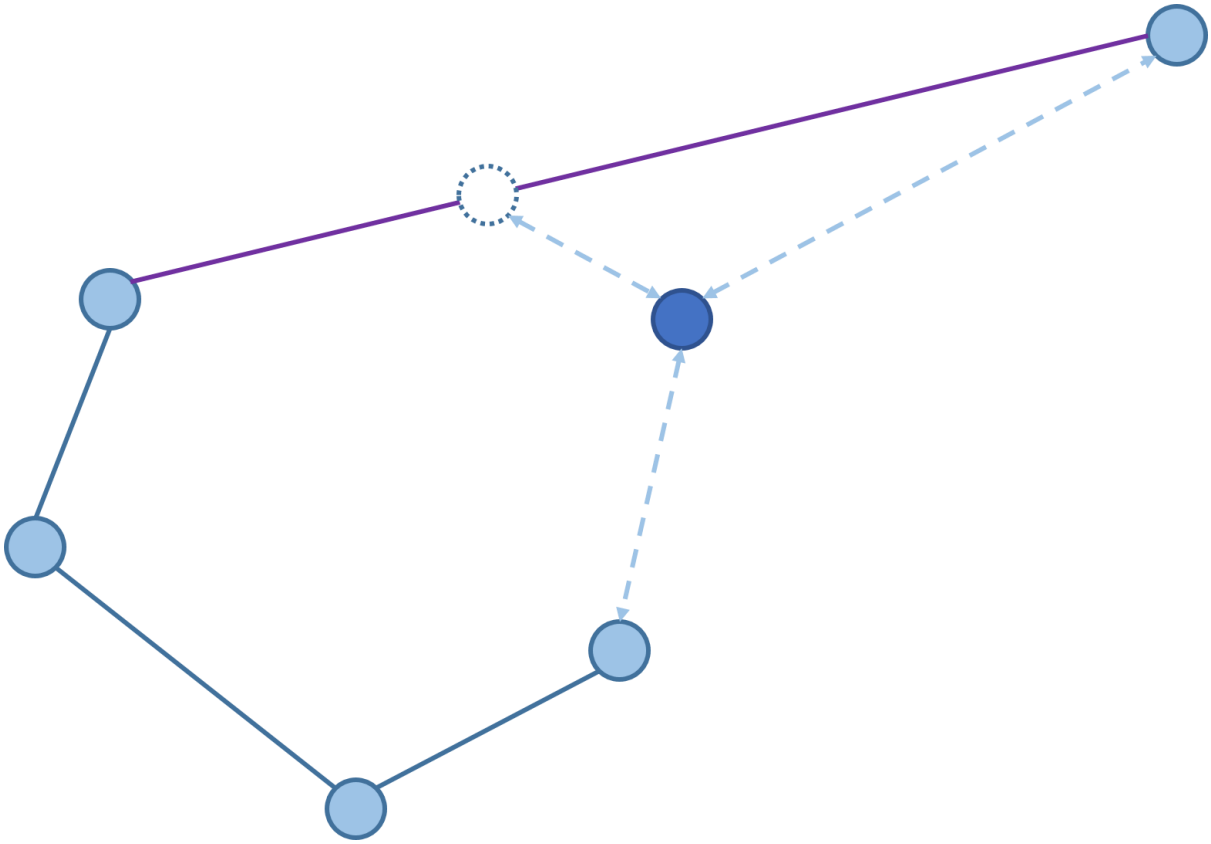

Figure 2

For a detailed information, refer below paper p.14-16.

[1] Karaman, Sertac, and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning." The International Journal of Robotics Research 30.7 (2011): 846-894.

Link: http://dspace.mit.edu/handle/1721.1/81442

Dynamic mapping

For dynamic path planning, you have to recognize the situation where environment is changed. So firstly you have to check whether environment around the robot obstructed or not by using kinect. Kinect provides depth information as point cloud data. Point cloud data has points (x, y, z) in 3D space. You cloud check the existence of an obstacle in front of robot from the data. If the path is obstructed, robot have to find a new path. Thus when your path are obstructed by dynamic environment you should draw a new map using point cloud data from Kinect and the transformation between world, robot and sensor. The transformation between Kinect frame, robot frame, world frame and grid map frame is represented as below figure 3.

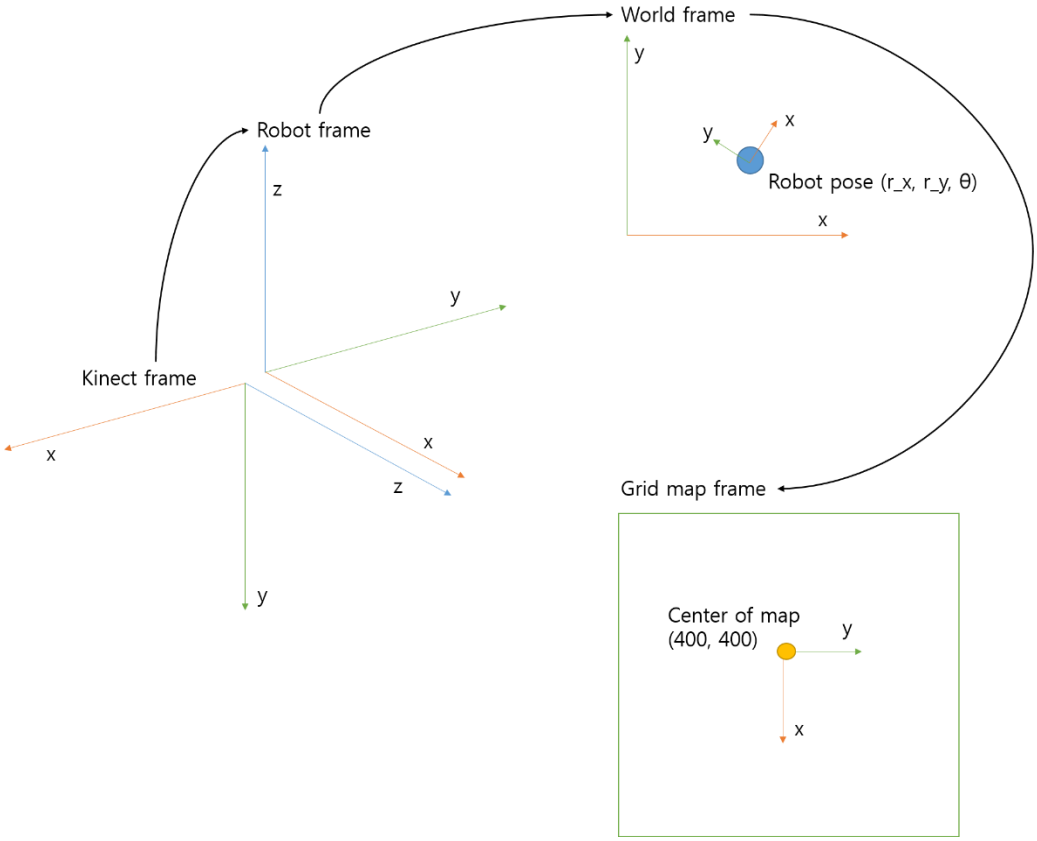

Figure 3

This procedure will be proceeded in PATH_PLANNING state. So when the robot meets the

unexpected obstacle state transition need to occur from RUNNING to PATH_PLANNING. State definitions are at the top of the main.cpp file. In many case, the localization step need to be proceeded before the mapping step. But you don't worry about localization problem since gazebo provide true position of robot. When you mapping the new environment robot's motion is quite important. The robot needs to look around since the kinect's sensing range is limited. Thus you made robot look around in PATH_PLANNING state. You could add some variable or function in main.cpp if you need for implementation.

TA's recommend: Design some threshold for state transition when obstacles emerge. And rotate the robot at fixed point using FSM during dynamic_mapping.

Skeleton Code

rrtTree.cpp

Fill in the **TODO** part in rrtTree.h and rrtTree.cpp

Main.cpp

Fill in the **TODO** part in main.cpp.

isCollison() is a function for checking obstacles around robot. isCollison() can be used for transition of state.

dynamic_mapping() is function for drawing a dynamic map. The subscriber for Kinect is predefined as variable "gazebo_kinect_sub" and it store the data from Kinect in variable "point_cloud". The "point_cloud" has the points as standard vector type. So data can be accessed by calling array operator such as following example.

```
for(int i = 0; i < point_cloud.size(); i++){
        point_cloud[i].x;
        point_cloud[i].y;
        point_cloud[i].z;
}
```

Scenario

We provide two scenario for checking rrt* algorithm and dynamic mapping. You could launch

original map using command "roslaunch project3 project3.launch" when testing your rrt* algorithm and changed map using command "roslaunch project3 project3_dynamic.launch" when testing your dynamic mapping algorithm.

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_03_[TeamName].tar.gz"**.