EECS 498 Final Project

Kinodynamic BiRRT with KD Tree Search

Sheng'ao Wang
wsashawn@umich.edu

December 19, 2020

Abstract

In this final project, I implemented a BiRRT algorithm from scratch to solve a non-holonomic planning problem. In order to increase the nearest neighbor searching process, a dynamically updating KD tree is used to store the configurations of the robot. A maze environment is created along with a hovercraft robot named "Wand" to test the algorithm. Regarding the complexity of this very maze, some modifications to the algorithm are made to better solve the planning problem.

1 Introduction

Rapidly-exploring Random Tree (RRT) has been one of the most essential sampling-based planning algorithms since developed by Steven M. LaValle and James J. Kuffner Jr. Comparing to exact algorithms, RRT holds its advantages in efficiently searching non-convex and high-dimensional spaces, making it perfectly suitable in a real environment full of irregular obstacles, especially when dealing with planning problem of robot arm.

Since most practical problems have non-holonomic constraints more or less, it's important to merge the consideration of non-holonomic constraints into the original RRT algorithm, making it so-called Kinodynamic RRT. By using simple primitives to explore configuration space, a path that satisfies non-holonomic constraints can be efficiently found without the need of modeling the whole environment. Thus, Kinodynamic RRT has been widely applied in area such mobile robotics to plan the trajectory in complex environment.

2 Implementation

2.1 Testing Environment Buildup

The environment that I played around is a flat maze where my Wand robot can wander and rotate horizontally. The structure of my maze is shown below, along with the start configuration and the goal configuration of Wand robot. The robot is considered arriving at the goal configuration

when the distances to goal in x and y direction are both less than 0.05, and the rotation angle difference is less than 10°.

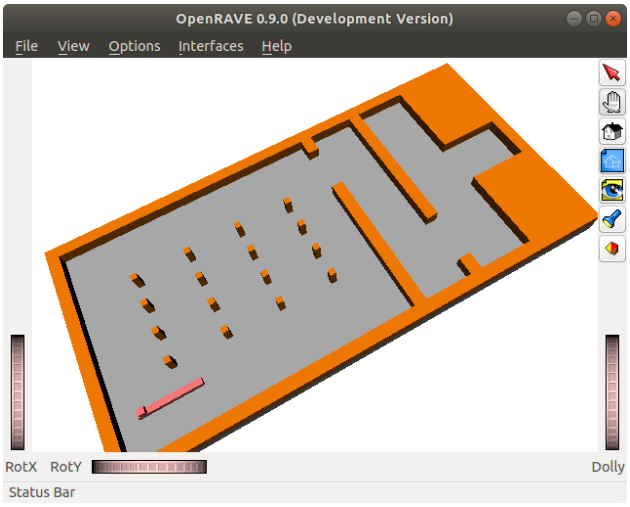

Figure 1: start configuration

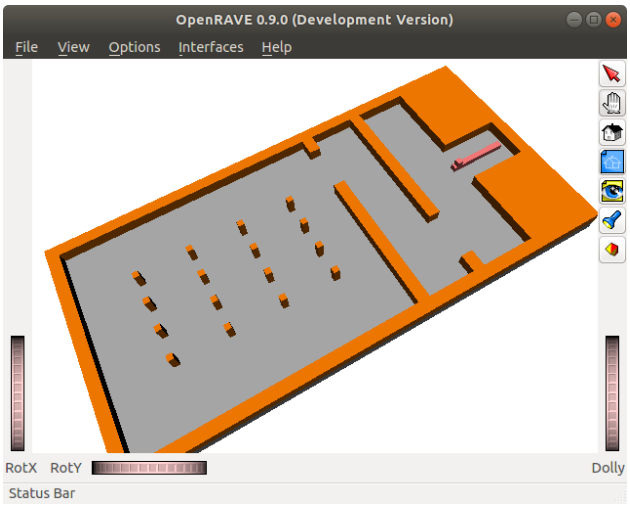

Figure 2: goal configuration

Note that the left part of the maze is full of sparse obstacles, leaving the robot some space to rotate but not that much space to move around. The right part of the maze is basically a narrow road, which further restrict the space for rotation. The difference between these two parts of maze is my primary motivation to use Bidirectional RRT (BiRRT) rather than a single RRT.

About the implementation detail, I created the environment by editing the xml file. The robot is created by using openrave built-in function, and its accelerations in (x, y, θ) directions are controllable.

2.2 BiRRT

The main heuristic of BiRRT is that growing two trees from start and goal configurations simultaneously would increase the exploring efficiency. Besides, I also leverage some other advantages of BiRRT in this particular planning problem, including but not limited to:

- **Avoid the difficulty of exploring around goal configuration.** As can be seen from the figure, the goal configuration is a pretty small region, thus cannot be reached from a large range of outer C-space. For example, even if the robot has arrived at the door, it may be still not able to enter the goal region since it holds a different rotation angle. Thus, by generating another tree from goal configuration, we can rapidly escape away from the small goal region and make the final connection between the former and latter parts of the path in a relatively larger region, which would definitely be easier than making the final connection around the small goal region.
- **Use difference setup in these two trees.** Sample method and extension type are important attributes for RRTs. Since different setup is useful in different environment structure, The planner can suit the environment better by deploying different RRT setup in different parts of the maze.

2.2.1 Main Algorithm

The main steps of BiRRT is shown in the diagram below. Note that since the trees are expanding towards each other rather than the goal configuration, the goal bias part is replaced by extending to the other tree's new point.

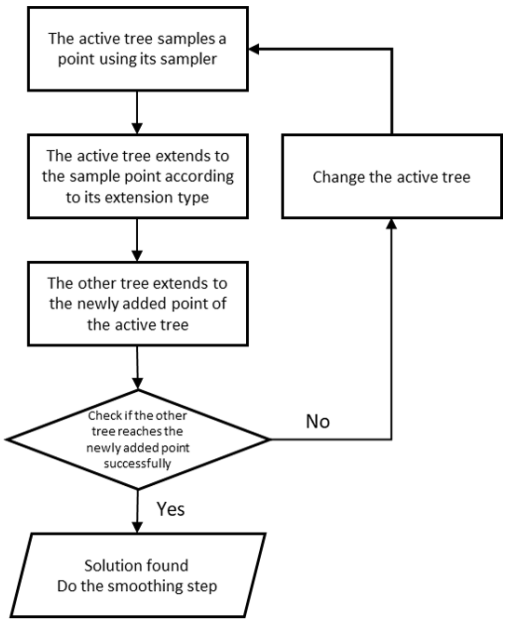

Figure 3: BiRRT algorithm

2.2.2 Distance Metric

Since the 3 DOFs of the robot (x, y, θ) have different units, different weights of different DOFs must be used while evaluation the distance between two points. I still use Euclidean distance as the basis distance, so the distance metric in this problem is:

$$D(q_1, q_2) = \sqrt{w_x(x_1 - x_2)^2 + w_y(y_1 - y_2)^2 + w_\theta(\theta_1 - \theta_2)^2} \tag{1}$$

where w_x, w_y, w_θ are the weights for these 3 DOFs.

2.2.3 Samplers

For the tree that grows from start configuration, I just used a simple uniform sampler around all the world.

For the tree that grows from goal configuration, since it grows in a narrow space, I used bridge method to sample (i.e., choosing the point in the middle of two near obstacles) in order to get more samples in the narrow roads rather than some other space with sparse obstacles.

2.2.4 Local Planners

The local planners should be easy enough to be fast executed whenever a tree needs to extend towards a target point. I use the idea of local search by discretizing the acceleration space into several primitives and do an 8-connect search to find the primitive that best approaches the target while not hitting the obstacles. Assume the robot is doing uniformly accelerated motion between every two planning step, then the calculation from primitive to next configuration is rather simple, given the current configuration and a fixed Δt. Note that the speed constraints must be observed.

$$q_{next} = q_{current} + v_{current}\Delta t + \frac{1}{2}a\Delta t^2 \tag{2}$$

Above is the process of extending one step towards the target point. Extending multi steps towards is just an iteration of one-step extend until reaching the goal or nowhere to extend. It worths mentioned that the local planner is allowed to extend away from the target point while doing one-step extend, i.e., the newly extended point is further from the target than the origin point, since exploring is always encouraged no matter if it's a "bad exploration". However, this tolerance must be canceled in multi-steps extend, otherwise it would lead to a dead loop of one-step extend.

2.2.5 Extension Type

For the tree that grows from start point, multi-steps extend is picked since it can increase the efficiency of exploring the world.

For the tree that grows from goal point, one-step extend is picked, since the robot is expected to explore carefully in that narrow space. However, a common result of multi-steps extend is that the

speed could increase dramatically, making it hard for the robot to stop before actually hitting the obstacles, i.e., those points near the obstacles often have no children points since all their children hit the obstacles heavily.

2.3 KD Tree

In the extending step of RRT, the planner needs to search for the nearest point among all the points that have been explored. If we don't use any trick and just traverse the whole point list whenever we take an extend step, the time complexity would be $O(n)$. Therefore, Some data structures such as KD-tree or Hash table can be used to store the points data and thus to reduce the time complexity.

In this project, I used KD tree to store the data. When the branching number is 2, KD tree is essentially a binary tree in which every leaf node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x value of the point, and its normal would be the unit x axis.

2.3.1 Search for the Nearest Point

Searching for a nearest neighbour in a k-d tree proceeds as follows. Its time complexity is $O(log(n))$, ideally.

- Starting with the root node, the algorithm moves down the tree recursively by comparing with the current node in the current split dimension.
- Once the algorithm reaches a leaf node, it checks that node point and if the distance is better, that node point is saved as the "current best".
- The algorithm unwinds the recursion of the tree, performing the following steps at each node:
  - If the current node is closer than the current best, then it becomes the current best.
  - The algorithm checks whether there could be any points on the other side of the splitting plane that are closer to the search point than the current best. In concept, this is done by comparing the current minimum distance with the distance to the split hyperplane.
    - If there could be nearer points on the other side of the plane, move down the other branch of the tree from the current node looking for closer points, following the same recursive process as the entire search.
    - Otherwise, continue walking up the tree.
- When the algorithm finishes this process for the root node, then the search is complete.

2.3.2 Insert a New Point

First, traverse the tree, starting from the root and moving to either the left or the right child depending on whether the point to be inserted is on the "left" or "right" side of the splitting plane. Once get to the node under which the child should be located, add the new point as either the left or right child of the leaf node, again depending on which side of the node's splitting plane contains the new node. This method's time complexity is also $O(log(n))$.

2.3.3 Balancing

With more and more points added to the KD tree, the tree could become unbalanced, i.e., the size of the left subtree and the right subtree is severely unequal. So we keep observing the ration between every left subtrees and their corresponding right subtrees. Once any ratio is over a threshold, collect all the points stored in this tree and reconstruct them. The construction proceeds as follow:

- For the given points, calculate the variance in each dimension, pick the biggest one as the split dimension.
- Sort these points by the split dimension, take the median as the root node.
  - If there are still points left, split the rest points into two lists, construct them by calling this method recursively.
  - Otherwise, the construction has ended, unwind the recursion.

2.4 Path Smoothing

One important shortcoming of sampling-based method is that the solution is often not optimal, thus we need to smooth the path after getting it from the RRT planner. Shortcut smoothing is used in this project, two passing points are randomly picked from the path and are transferred to local planner to see if they can be directly connected. Here we use the same local planner as we use in RRT extend step.

3 Results

3.1 Robot Setup

The constraints applied to the robot, such as velocity constraints or control frequency, can heavily influence the result of the experiment. At first, I set the control frequency to 100HZ just like what's common in real mobile robot, resulting in a delicate trajectory with a desperately long runtime. So I reset the control frequency to 10Hz, the efficiency got increased a lot indeed, but sometimes the result trajectory can penetrate the obstacles in an impossible way, like what is shown below. This is due to my former collision checker's missing, since it only samples checks few points between each two connecting points, the robot may cheat on the collision checker by speeding up

and hit up the obstacle straightly. This phenomenon is named "Quantum Tunneling of Wand" by future Dr. Shawn Wang.

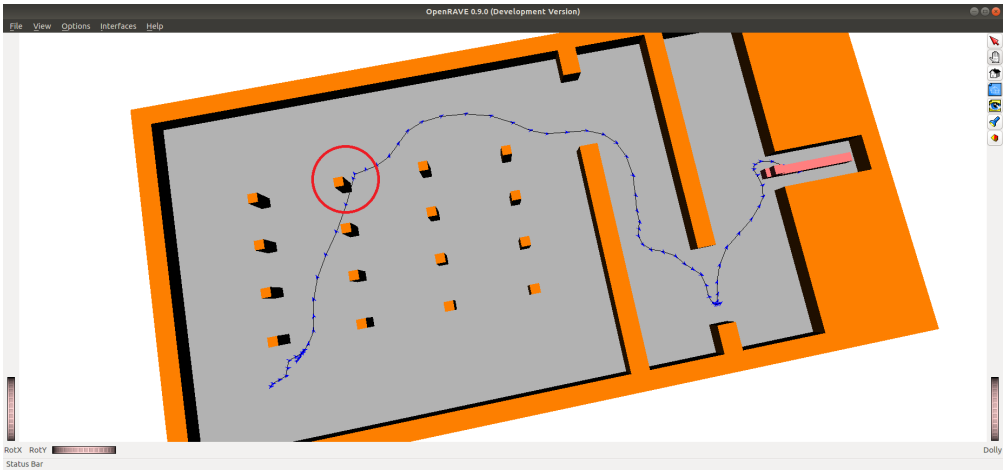

Figure 4: obstacle penetration occurs in red circle

Anyway, I solve it by adding more check points to the collision checker and tightening the velocity constraint. Another advantage of tightening the velocity constraint is that it offers the planner more chances to explore the unknown space since all the movements have become more conservative and are less likely to have no children points alive.

3.2 Primitive Number Selection

In order to observe the influence of primitive number to my BiRRT planner's performance, I setup three sets of experiments with primitive numbers 3,4,5 respectively, everything else remain the same. Each set of experiments contains 3 independent experiments with random seeds from 0 to 2 respectively. The results are shown below:

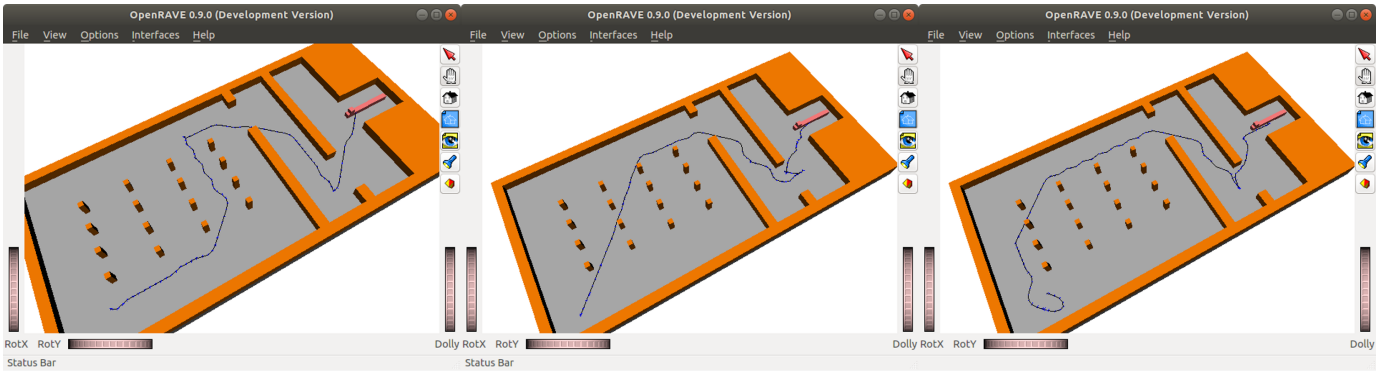

Figure 5: path with 3 primitives

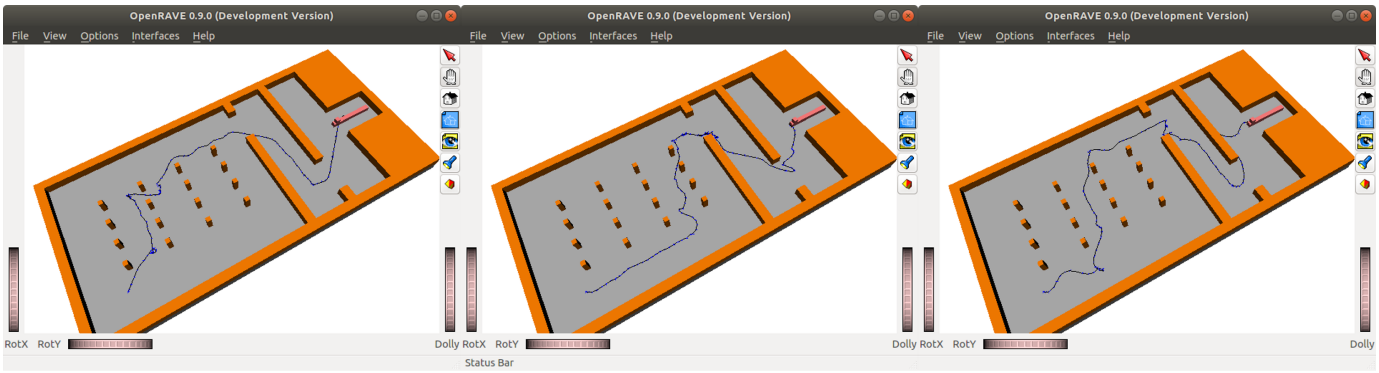

Figure 6: path with 4 primitives

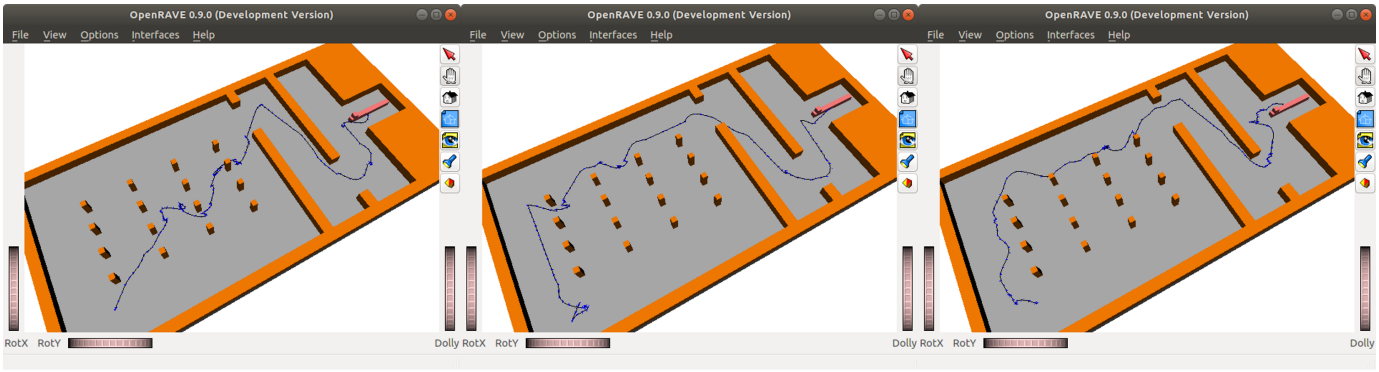

Figure 7: path with 5 primitives

The computation time and the path length (before and after smoothing) for the different number of primitives is shown in Table 1 and Table 2 below, respectively.

Number of Primitives	Experiment 1	Experiment 2	Experiment 3
3	93	398	66
4	61	224	145
5	220	115	183

Table 1: computation time (second)

Number of Primitives	Experiment 1	Experiment 2	Experiment 3
3	244 $\longrightarrow$ 108	233 $\longrightarrow$ 120	178 $\longrightarrow$ 126
4	202 $\longrightarrow$ 133	263 $\longrightarrow$ 176	274 $\longrightarrow$ 164
5	229 $\longrightarrow$ 188	292 $\longrightarrow$ 210	278 $\longrightarrow$ 204

Table 2: path length before and after smoothing

I've expected that using more primitives would bring me a better path at a cost of increasing computation time. However, according to the path length and the path shown in the figure, although using more primitives does increase the computation time, it nevertheless decreases the quality of the path. Combining with my observation of RRT exploring process, my guess is that with more primitives to choose, RRT has more chance to explore the unknown space, thus the chance of forming a tree with weird shape is increased, making the final path twist a lot.

4 Conclusions and Future Works

In this project, I designed a maze environment with different geometry features in its different parts. In order to fully utilize the special features of this maze, I implemented a BiRRT planner and assigned different setup to different tree. The experiments have shown the ability of this planner, as well as revealed some interesting performance changes when the primitive number changes.

Due to time limit, the following are my ideas that have been thought about but haven't got the chance to be implemented:

- Set up more experiments with different RRT planner configuration to compare them in different viewpoint.
- Implement k-nearest search and pick all the k nearest points randomly with some probability. This may be able to help a planner out when it gets its newest points stuck in some corner.
- Divide C-space into several grids, at each sample step, first randomly pick one grid with some probability, then do a further sample in that grid. The probability of a grid being picked should have negative correlation with the number of points inside it. This can encourage the planner to explore the unknown space more.
- Merge the consideration of velocity difference in the formulation of distance matric. This can make the trajectory more smooth.