RRT* Quick- A motion Planning Algorithm with Faster Convergence Rate

Paras Savnani
Robotics Engineering, ENPM
University of Maryland
College Park, USA
psavnani@umd.edu

Pooja Kabra
Robotics Engineering, ENPM
University of Maryland
College Park, USA
pkabra@terpmail.umd.edu

***Abstract*—In recent times, sampling-based algorithms have been employed extensively in a high dimensional space and are also shown to work well. Besides they are proven to be probabilistically complete and asymptotically optimal. One such algorithm that has recently become abundantly popular in the path planning fraternity is Rapidly-exploring Random Tree star (RRT*). In this project, we explore RRT*-Quick as an improved version of RRT*. RRT*-Quick embarks on the observation that the nodes in a local region have common parents. It is shown to have a faster convergence rate without being computationally inefficient by using ancestor nodes to grow the reservoir of candidate parent nodes.**

***Keywords - Sampling-based algorithms,* probabilistically complete, asymptotically optimal, *RRT, RRT*,* RRT*-Quick**

I. Introduction

Conventional graph-based planning methods require us to discretize the search space. While they are complete and perform remarkably till 5 dimensions, as the dimension of the search space increases, the number of cells and hence computational burden grows exponentially. The solution becomes more and more difficult to reach. This complexity makes them unfavourable for practical uses.

Random sampling based algorithms are online algorithms. They null the need to have a graph before we start the path planning problem. The idea is to randomly sample nodes in the search space and keep connecting them to the existing tree, and return a solution as soon as it exists. The possible solution is reached quickly and further time is used by the algorithm to optimize the solution. As the obstacle space now does not have to be modelled in the state space, these algorithms hugely save computational costs. Although theoretically not complete, they assume probabilistic completeness as the number of nodes tend to infinity.

One of the more commonly used sampling based-algorithms is RRT. RRT* is an advanced variant of RRT. While RRT only provides a feasible solution, RRT* improves the solution as iterations progress.

In this project, we present RRT*-Quick, an extension of RRT* that causes faster convergence by hinging on the characteristic that neighboring nodes have common parent nodes.

The report is laid out in the following order: Section XX introduces the vanilla RRT and RRT* algorithms. RRT*-Quick is described in the background section. Results from simulations in Pygame and Gazebo environment follow in Section XX. We finish with concluding comments.

II. Background

This section states the basic RRT and RRT* algorithms.

A. RRT

RRT works by iteratively sampling a random node in the state space and connecting a new node in that direction from a random node in our tree. RRT is probabilistically complete and will almost definitely give you a solution when the search space becomes dense enough. However, it only finds us a solution and it might not necessarily be optimal. In fact, it is rarely optimal. Below is the pseudocode for RRT:

Input: $T = (V, E)$
Output: T
while $i < N$ **do**
  $x_{rand} \leftarrow Sample(i)$;
  $x_{nearest} \leftarrow Nearest(T, x_{rand})$;
  $x_{new} \leftarrow Extend(x_{nearest}, x_{rand})$;
  $\sigma_{new} \leftarrow Steer(x_{nearest}, x_{new})$;
  if $CollisionFree(\sigma_{new})$ **then**
    $V \leftarrow V \cup x_{new}$;
    $E \leftarrow E \cup (x_{nearest}, x_{new})$;
  end
end
$T \leftarrow (V, E)$;
return T;

Fig. 1. RRT Pseudocode

B. RRT*

RRT* builds on the previous RRT algorithm. Once a

Input: $T = (V, E)$
Output: T
while $i < N$ **do**
  $x_{rand} \leftarrow Sample(i)$;
  $x_{nearest} \leftarrow Nearest(T, x_{rand})$;
  $x_{new} \leftarrow Extend(x_{nearest}, x_{rand})$;
  $\sigma_{new} \leftarrow Steer(x_{nearest}, x_{rand})$;
  if $CollisionFree(\sigma_{new})$ **then**
    $X_{near} \leftarrow Near(T, x_{new})$;
    $x_{min} \leftarrow x_{nearest}$;
    $c_{min} \leftarrow Cost(x_{nearest}) + Cost(\sigma_{new})$;
    foreach $x_{near} \in X_{near}$ **do**
      $\sigma_{near} \leftarrow Steer(x_{near}, x_{new})$;
      if $Cost(x_{near}) + Cost(\sigma_{near}) < c_{min} \wedge CollisionFree(\sigma_{near})$ **then**
        $x_{min} \leftarrow x_{near}$;
        $c_{min} \leftarrow Cost(x_{near}) + Cost(\sigma_{near})$;
      end
    end
    $V \leftarrow V \cup x_{new}$;
    $E \leftarrow E \cup \{(x_{min}, x_{new})\}$;
    foreach $x_{near} \in X_{near}$ **do**
      $\sigma_{near} \leftarrow Steer(x_{new}, x_{near})$;
      if $Cost(x_{new}) + Cost(\sigma_{near}) < Cost(x_{near}) \wedge CollisionFree(\sigma_{near})$ **then**
        $E \leftarrow E \setminus \{(Parent(x_{near}), x_{near})\}$;
        $E \leftarrow E \cup \{(x_{new}, x_{near})\}$;
      end
    end
  end
end
$T \leftarrow (V, E)$;
return T;

Fig.2 . RRT Pseudocode

new node is connected to the existent tree, RRT* considers all its neighboring nodes in a volume sphere of a certain radius around the new node. The radius of the sphere *r* is given as:

$$r = \gamma\left(\frac{log(n)}{n}\right)^{\frac{1}{d}}$$

γ is the planning constant,

n is the number of nodes,

d is the dimension of the state-space

Cost to get to the new node from each of them is then computed and the new node is rewired to the one that results in the lowest cost. This is the *choose parent* step. Once the new parent is chosen, it is time to *rewire* the neighbors. If routing any of the neighbors via the new node results in an even lower cost for them than they already are at, the new node is made their parent. Unlike RRT, the RRT* algorithm returns an increasingly optimal path as the iterations multiply. In other words, RRT* is asymptotically optimal. The figure above shows the pseudocode for RRT*.

Two techniques often combined with RRT* are Committed Trajectory and Branch-and-Bound.

Committed Trajectory: The algorithm splits the optimization problem into a committed trajectory and an uncommitted tree. The early part of the current most optimal path is frozen and it becomes the committed trajectory. Its last node is now treated as the root of the uncommitted tree. Our algorithm will now sprout new paths from the root that form our uncommitted tree, which will yield better and better solution as it grows dense.

Branch-and-Bound: This technique clips off excess nodes from the tree. Once we find a feasible path to the goal node, we use its cost as an upper limit. All nodes that might have greater cost and all their ancestors are removed from the tree. This will result in lesser computation time for each iteration as there will be lesser neighboring nodes to process every time.

III. Methodology

We start by implementing the vanilla RRT and RRT* and visualizing them in the pygame environment. Then, we build the RRT*-Quick over the base RRT* version by modifying the choose parent step and the rewiring function. We also write the rewire function so as to consider the ancestors of a node to a certain degree.

A. Implementation of the Algorithm

The RRT*-Quick class inherits from the RRT and RRT* classes. The planning function is the main function which is responsible for driving the algorithm. Initially, a random node is sampled in the obstacle space and the nearest node is found. Then, the nearest node is steered towards the random node. After this, the check collision function is used to check if the new node lies in the obstacle space or not. This function implements the half-plane equations on the obstacle space and returns True if the node is not in obstacle space and returns false in the other case. After this step the near nodes are found using the distance metric and their corresponding ancestors are found. We take the union of these near nodes and the ancestor nodes and we choose the parent of the new node based on the least distance criteria. After choosing the parent we rewire the near nodes based on the ancestor nodes and we append the new node to the node list.

In the *choose parent* function we iterate over all the near nodes and steer each near node towards the new node and we return assign then node with the least cost as the parent node of the new node.

In the *rewire* step we iterate over all near nodes and in each iteration we iterate over ancestor nodes of the new node and steer the ancestor nodes to find the least cost path and assign it to the near node. The important step in rewiring is the propagation of the costs to the leaves of the new node, this is a recursive function which helps to update the costs of all the children nodes.

The degree of the ancestors to use determines the path straightness, ideally we should use 3-4 ancestors but we can use infinite degree as it won't matter after a certain limit.

Figure 3 shows the triangle inequality condition in this algorithm. From the example it is evident that the near node has a path from the start node with a cost of 24 units and via the new node it has a cost of 18 units. So RRT* will rewire the new node with the near node, but the RRT* Quick will rewire the ancestors with the near node as they will have lesser costs compared to the new node.

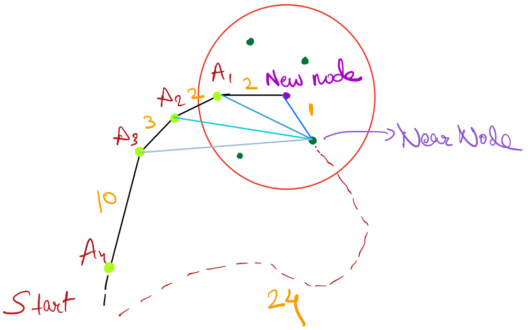

Fig. 3. Triangle Inequality Condition

Figure 4 shows the pseudocode of the RRT* Quick algorithm, it is similar to the RRT* where it has choose parent and rewire steps. But, it differs in using the ancestor nodes to optimize the path. Now, the path can be optimized by increasing the radius of comparison for near nodes or by increasing the ancestors of the new nodes. And this algorithm claims to improve the cost with almost same time complexity of the RRT*. Therefore, it is better with respect to the other sampling based algorithms.

Input: $T = (V, E)$
Output: T
while $i < N$ **do**
  $x_{rand} \leftarrow Sample(i)$;
  $x_{nearest} \leftarrow Nearest(T, x_{rand})$;
  $x_{new} \leftarrow Extend(x_{nearest}, x_{rand})$;
  $\sigma_{new} \leftarrow Steer(x_{nearest}, x_{rand})$;
  if $CollisionFree(\sigma_{new})$ **then**
    $X_{near} \leftarrow Near(T, x_{new})$;
    $x_{min} \leftarrow x_{nearest}$;
    $c_{min} \leftarrow Cost(x_{nearest}) + Cost(\sigma_{new})$;
    foreach $x_{near} \in X_{near} \cup Ancestors(X_{near}, degree)$ **do**
      $\sigma_{near} \leftarrow Steer(x_{near}, x_{new})$;
      if $Cost(x_{near}) + Cost(\sigma_{near}) < c_{min} \wedge CollisionFree(\sigma_{near})$ **then**
        $x_{min} \leftarrow x_{near}$;
        $c_{min} \leftarrow Cost(x_{near}) + Cost(\sigma_{near})$;
      end
    end
    $V \leftarrow V \cup x_{new}$;
    $E \leftarrow E \cup \{(x_{min}, x_{new})\}$;
    foreach $x_{near} \in X_{near}$ **do**
      $\sigma_{near} \leftarrow Steer(x_{new}, x_{near})$;
      $X_{candidates} \leftarrow Ancestors(x_{new}, degree) \setminus Ancestors(x_{near}, degree)$;
      $x_{min} \leftarrow x_{new}$;
      $c_{min} \leftarrow Cost(x_{new}) + Cost(\sigma_{new})$;
      foreach $x_c \in X_{candidates}$ **do**
        $\sigma_c \leftarrow Steer(x_c, x_{near})$;
        if $Cost(x_c) + Cost(\sigma_c) < c_{min}) \wedge CollisionFree(\sigma_c)$ **then**
          $x_{min} \leftarrow x_c$;
          $c_{min} \leftarrow Cost(x_c) + Cost(\sigma_c)$;
        end
      end
      $E \leftarrow E \setminus \{(Parent(x_{near}), x_{near})\}$;
      $E \leftarrow E \cup \{(x_{min}, x_{near})\}$;
    end
  end
end
$T \leftarrow (V, E)$;
return T;

Fig. 4. RRT Quick Pseudocode*

IV. SIMULATION

The algorithms are visualized using pixel discretization in a convex polygon obstacle space. The obstacle space is a 10*10 grid with polygon obstacles. Obstacles can be made more complex, but they are kept simple for the simulation to visualize the planning algorithm.

A. Environment

For visualization, we use *Pygame (3rd Party Python Dependency)* to generate the obstacle space and the path. Pygame uses an infinite while loop to control the animation within the window. We implement this loop in the planning function for max iterations. We generate the obstacle space using the Pygame draw functions to draw the polygons and update them in each iteration. Also, obstacles are inflated to give clearance to the robot.

After finding the shortest path we iterate through the nodes in it and circles for the x,y coordinates of the node.

B. Results

To compare the three algorithms we calculate each algorithm's cost for the start node to the goal node for different iterations. As these are sampling based algorithms, the costs depend upon sampling, but we can get a general idea of the algorithm's optimality with the comparison.

It was observed that the cost of RRT does not decrease with increasing the number of iterations, because there is no cost consideration and rewiring in RRT, so it stops after finding a path. But, for RRT* there are two important steps choosing the parent and re-wiring, so it improves the cost of the near nodes in each iteration. It considers a circle of a given radius and rewires the new node based on the near nodes. Furthermore, for the RRT* Quick algorithm it considers the union of ancestor nodes and the near nodes in the choose parent step and also rewires all the near nodes in the rewiring step with respect to the ancestor, so it gives much lower costs and more straight paths for the same number of iterations.

TABLE.1.COST COMPARISON OF ALGORITHMS

Iterations	Cost Comparison Of Algorithms (Euclidean Distance)		
	RRT	***RRT****	***RRT* Quick***
1000	13.60	12.25	13.23
2000	13.40	11.90	12.48
3000	14.60	11.80	11.66

Table 1 shows the comparative costs of the algorithms for 1000, 2000, and 3000 iterations. It is evident that as we increase the number of iterations the RRT*-Quick starts outperforming the other two algorithms. For 3000 iterations it has the least cost of 11.66 and if we further increase the number of iterations or the ancestor hierarchy, it will improve further. On the other hand RRT* is not far behind for small iterations it is almost as good as the Quick algorithm. But RRT is not comparable to these algorithms, because it has no cost tracking and rewiring.

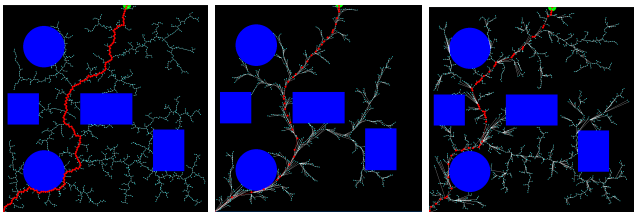

Fig. 3. RRT vs RRT vs RRT*-Quick (2000 iterations)*

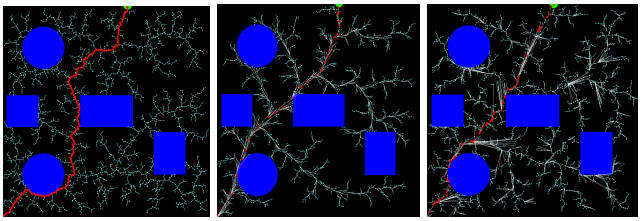

Fig. 4. RRT vs RRT vs RRT*-Quick (3000 iterations)*

C. Gazebo Simulation

This project is simulated in a gazebo world. The world is shown in figure 5. It is a 10 x 10 grid with polygon obstacles.
ROS provides the capability of creating publishers and subscribers. We use two ROS nodes to achieve the simulation. The planner node contain the RRT/RRT*/RRT* Quick algorithms which writes the shortest path to a text file. This file is read by the robot_control node to drive the turtlebot.

Robot Control uses sensor msgs to detect and avoid the obstacles. A subscriber is used to receive laser scan msgs and these are divided into eight sections. The robot is only allowed to move if the front 45 degrees are obstacle free. Furthemore, it receives the position and rotation of the robot from the odom frame and uses geometry msgs to publish velocities to move the TurtleBot. A proportional closed loop

controller is implemented with linear and angular gains to drive the robot in the environment.

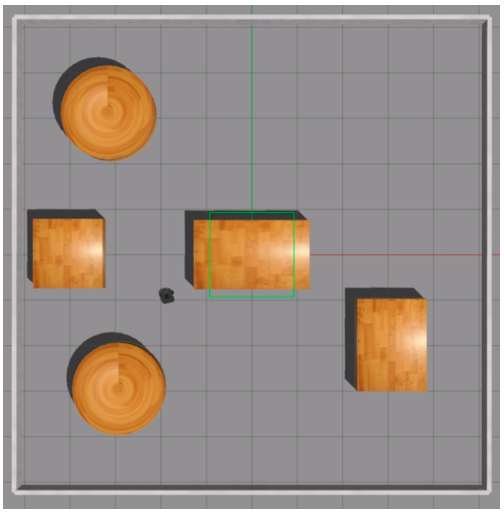

Fig. 5. Gazebo Simulation

Conclusion and Future Work

We implement RRT*-Quick algorithm in this project. It has a higher convergence rate compared to the RRT* with almost similar time complexity. It leverages the ancestor nodes of the near nodes to rewire the existing paths in the graphs. It prunes the path on each iteration to make it straight and thus reduces the overall cost. For future work we aim to improve this algorithm to have even lesser costs by including the cost-to-go heuristic with this algorithm. Also, real world implementation with TurtleBot using ROS platform will be carried out.

References

[1] n-Bae Jeong, Seung-Jae Lee, and Jong-Hwan Kim- RRT*-Quick: A Motion Planning Algorithm with Faster Convergence Rate Robot Intelligence Technology and Applications 3 pp 67-76.

[2] Sertac Karaman, Emilio Frazzoli- Incremental Sampling Based Methods for Optimal motion Planning

[3] Karaman, Sertac et al. Anytime Motion Planning using the RRT ."2011 IEEE International Conference on Robotics and Automation(ICRA) May 9-13, 2011, Shanghai International Conference Center, Shanghai, China.