Sampling-Based Path Planning Using RRT Algorithm with Visualization Using Pygame*

Hemanth Joseph Raj
117518955
University of Maryland
College Park, MD, USA
hemanth1@umd.edu

Venkata Sri Ranga Sai Anapu
118529950
University of Maryland
College Park, MD, USA
vanapu@umd.edu

***Abstract*— In any world if we know the map of the world, we can, through the use of any path planning algorithm, arrive at a path based either on optimality, cost, time etc. Of the two major types of, the probabilistic based path planning algorithms, the Rapidly Exploring Random Tree (RRT), it's derivative the RRT* performs slightly better than its Roadmap counterparts since the roadmap algorithms like Probabilistic Road Maps (PRMs) focus more on building the graph than generating a path for the graph. Being an efficient algorithm, the RRT* builds upon the RRT algorithm by tweaking the tree as it is generated to always go back to the start with the least cost. This ensures a least cost path is always the output. This least cost path can be more optimized by lowering the threshold radius at which the new node will placed. But this is a tradeoff against faster output generation. (A variation of the RRT*, namely the goal-biased RRT* is also explored)**

Keywords*— *path planning, autonomous robots, Rapidly Exploring Random Trees, RRT*, sampling-based approach, graph generation style

I. INTRODUCTION

The defining aspect of any autonomous vehicle is its ability to operate autonomously in the world. Right from self-realization, more popularly known as localization; getting an understanding of how the environment looks like, where are the free spaces and where the obstacles lie; using the above two pieces of information to generate a path to the goal position; usage of sensors and controllers to guide the system along the path avoiding obstacles all by itself with no human intervention. This is still a huge problem and area of research in the robotics industry. Many automobile companies are in the race to bring out the first self-driving car, an implementation of the "autonomous system" used for transportation. As hypothesized by Paul Bourgine and Francisco J Varela, an autonomous system is *operationally closed*, meaning no input is given manually to control the system. They draw a stark difference between a **hetero**nomous system whose input/output is mapped by means of a set of instructions and an **auto**nomous system whose behaviors and interactions result from "self-organizing processes" [1].

A mobile robot is just another instance of an autonomous systems. Other such autonomous systems include, autonomous drones, guidance-controlled rockets, industrial manipulators are among the most popular types of an autonomous system. Robots these days are majorly used in all kinds of scenarios mostly to automate the most mundane of tasks. There are widely used in areas which aren't accessible by humans like Urban Search and rescue in a disaster struck region or in places where it not advisable for humans to venture into, like nuclear facilities, underground mines etc. After the 1986 Chernobyl nuclear reactor incident, several USSR and German designed mobile robots were deployed to help with cleaning up the radioactive waste materials [2]. This incident along fast-tracked the research and necessity for better multi-purposed mobile robots, preferably with autonomous capabilities.

To develop these autonomous capabilities and to be deemed to be considered "operationally closed" [1], the robot must have certain attributes as mentioned previously. One of these capabilities is – provided a map of the area, either through inputs or through SLAM by the usage of sensors, to arrive at a solution path for the robot to travel along so as to reach the assigned goal position from the start position. This is where path planning makes its introduction. Path Planning, sometimes referred to as Motion planning, is a computational problem whose aim is to find a sequence of valid configurations that moves the object (in our case a robot) from the initial state to the goal state. Path Planning is used in many areas of science such as computational geometry, computer animation, robotics, computer games etc. In a broader sense, the planning algorithms are classified into a.) Systematic Methods and b.) Probabilistic Methods [3]. They differ in the way the graph is generated in the map. The systematic methods take a defined action set to produce a new set of outputs and this pattern is applied at each step. In probabilistic methods, the new position is generated by the use of random function to generate a position anywhere in the map. The probabilistic method is also called as the Sampling-based method for path planning. Sampling-based path planning is considered to be "probabilistically complete" which refers to the fact that given a long enough run time, the algorithms will return an output containing the solution path, provided one exists [4]. The sampling-based methods are further classified into Probabilistic Road Map (PRM) and Rapidly Exploring Random Tree (RRT). In PRM, you take the given workspace, and configurations are randomly sampled by picking coordinated at random. These sampled configurations are then check if they collide with the obstacle space, if they do, they are eliminated. The retained configurations which aren't in obstacle space are called milestones and each of them are linked to their nearest neighbors by a straight line and collision free paths retained which gives you a map between all points in the free space. Between these milestones the configuration space is searched and the path is obtained. This is a brief explanation of the PRM.

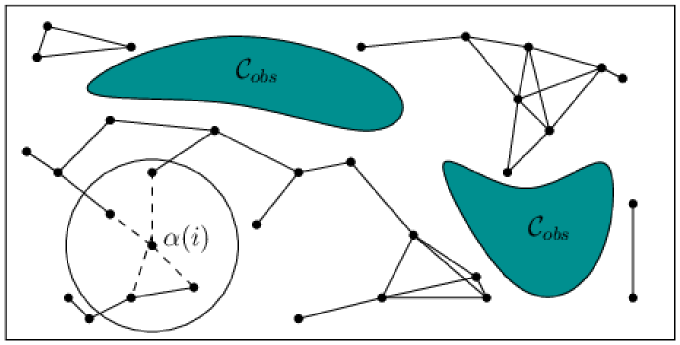

Figure 1: The sampling-based roadmap of PRM

(Credits: Steven M LaValle) [11]

The second type of sampling-based methods are called as Rapidly Exploring Random Trees (RRTs). This too has configurations which are randomly sampled. There are many variations of the RRTs such as RRT* which uses a heuristic to get optimal paths, Informed RRT* etc. In this implementation the RRT* algorithm is being implemented to generate the path. Its operations are explained in exhaustively in this paper.

This algorithm is being implemented using Python. Visualization is being done by the use of the Pygame library from Python since it makes the visualization easier to view.

This report is organized as follows. The current section *Introduction* contains the background information about autonomous systems, mobile robots, the need for mobile robots. Then it delves into what aspects makes a system autonomous and the it discusses about different path planning approaches, a brief description of probabilistic roadmaps. In the next section, *Previous Studies* it briefly discusses about various approaches prevalent in academia towards the usage of the RRT* algorithm. The report first states the approach by the authors of the paper on which this implementation is based on. And then a few other approaches are discussed

II. PREVIOUS STUDIES

In [5], Gurel et al approach the experimental results of the RRT* algorithm is being implemented on a real robot to autonomously travel around the real world. The algorithm was implemented for both static and dynamic obstacles. The map is generated by performing SLAM by making the use of Turtlebot3 Waffle_Pi which has an inbuilt Raspberry pi camera. The robot is moved along the world by using Teleop to capture the world, perform SLAM and output a map of the environment. This map is then used to create the path by applying the RRT* algorithm to it. A simulation too was done by the use of Robot Operating System and Gazebo on the Turtlebot3. The paper ends with the future work that needs to be undertaken to accelerate the convergence rate of the developed RRT* algorithm. In [6], Chang-an et al. had their RRT algorithm simulated and tested in an unknown world and compared it against an RRT algorithm they themselves developed. In [7], Moon and Chung addressed the problem of generating real-time trajectory for two-wheeled mobile robots with differential drives keeping in mind the non-holonomic and dynamic constraints.

III. METHOD

A. *Introduction of RRT and RRT**

Rapidly Exploring Random Tree (RRT) is a sampling-based path planning method. In other systematic planning algorithms, the action is usually predefined as per the holonomic or non-holonomic constraints of the system. In RRT, the new node is decided by randomly sampling the entire space. Hence it gets the term *Random Tree*. Since it randomly samples the entire space, the tree starts to *rapidly explore* the entire free space. The RRT algorithm was developed by Steven M. LaValle in 1998 [8] and further developed with the help of James J. Kuffner Jr. in 2001 [9]. Since then, many variations of the RRT algorithm have come up which build upon the RRT algorithm and apply other heuristic functions to optimize their solutions as per their needs.

A variation of the Rapidly Exploring Random Tree algorithm is the RRT* algorithm. RRT*, popularized by Dr. Karaman and Dr. Frazzoli [10], is an optimized modified algorithm that aims to achieve a shortest path, whether by distance or other metrics. The rapidly incremental nature of the algorithm helps it to uphold the kinematic constraints. It can also be used to create path planning keeping into consideration the kino-dynamic and non-holonomic constraints. With high speed of execution, the path planners based on RRT* tend to produce quicker results as compared to other algorithm which may take time to produce results.

The RRT* algorithm improved upon the RRT algorithm by two simple steps.

a. Once a new node is found, look through the nodes and find the best parent to it which has the least cost back to the start node

b. Rewire the path so that the path reflects the least cost back to start node

These above two changes help produce optimal paths between start node and goal node. It has the best of both worlds. It is computationally quicker because it uses the RRT algorithm as its base and since it always checks least cost back to the start node, the final path produced has lower cost as compared to the path produced just by RRT algorithm.

The RRT algorithm is run over a particular number of iterations and with each iteration, the possibility to reaching the goal state is massively increased since the tree keeps growing more in unexplored spaces producing faster convergence.

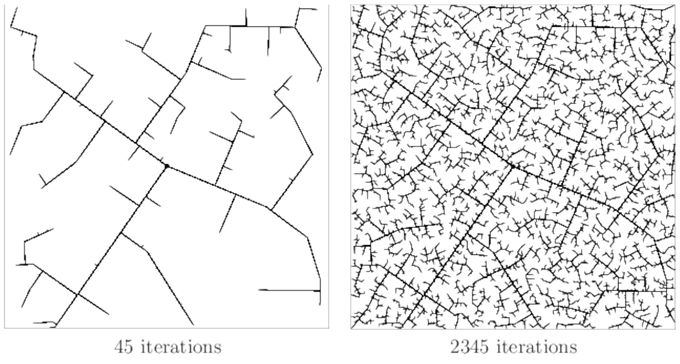

Figure 2: The tree generation for RRT algorithm for different number of iterations

(Credits: Steven M LaValle)

As it can be seen it the above figures that by increasing the number of iterations i.e., the number of times the RRT algorithm is run, the map keeps getting denser. This results in almost the total areas of the configuration space being explored.

B. *Pseudo Code of the Algorithms*

The RRT* algorithm can be easily defined done by three distinct algorithms as seen below

The first algorithm => The Main loop

```
Algorithm 1: $T = (V, E)$ ←RRT*($q_{init}$)
1 $T$ ←InitializeTree()
2 $T$ ←InsertNode($\emptyset$,$q_{init}$,$T$)
3 for $k \leftarrow 1$ to $N$ do
4     $q_{rand}$ ←RandomSample($k$)
5     $q_{nearest}$ ←NearestNeighbor($q_{rand}$,$Q_{near}$,$T$)
6     $q_{min}$ ←ChooseParent($q_{rand}$,$Q_{near}$,$q_{nearest}$, Δq)
7     $T$ ←InsertNode($q_{min}$, $q_{rand}$, $T$)
8     $T$ ←Rewire($T$, $Q_{near}$, $q_{min}$, $q_{rand}$)
9 end
```

Figure 3: Pseudo Code 1 – Main loop

(Credits: Devin Connell & Hung Manh La) [12]

As we can see in Algorithm 1 above that we first initialize the tree. Get the start node q _ init and add it to the Data Structure. Now here N refers to the number of iterations the algorithm will be run for so as to get a fine solution. For each iteration a random sample configuration is generated in the free space. Then the nearest neighbors check it done. This is done by the use of the radius of search function from the new node which makes a list of all neighbors in the vicinity of the radius.

Now the next algorithm => ChooseParent

As we can in this algorithm, first the nearest neighbor is selected and its cost is estimated. Then we iterate through the list of all the nearest neighbors and generate the path, if the path doesn't seem to be in any obstacle space proceed ahead. Now we update the cost for that particular neighbor and then the check it against the minimum cost. If it is lesser, then the new cost becomes the minimum cost and that node is designated as the nearest node with least cost back to start. This operation is performed for all nodes in the nearest neighbor list. What this ends up doing is iteratively choose that node with the least cost of all within the vicinity radius as the parent node and returns that value. This is saved in the nodes list.

```
Algorithm 2: $q_{min}$ ←ChooseParent($q_{rand}$,$Q_{near}$,$q_{nearest}$,Δq)
1 $q_{min} \leftarrow q_{nearest}$
2 $c_{min}$ ←Cost($q_{nearest}$) + c($q_{rand}$)
3 for $q_{near} \in Q_{near}$ do
4     $q_{path}$ ←Steer($q_{near}$, $q_{rand}$, Δq)
5     if ObstacleFree($q_{path}$) then
6         $c_{new}$ ←Cost($q_{near}$) + c($q_{path}$)
7         if $c_{new} < c_{min}$ then
8             $c_{min} \leftarrow c_{new}$
9             $q_{min} \leftarrow q_{near}$
10        end
11    end
12 end
13 return $q_{min}$
```

Figure 4: Pseudo Code 2 – Choosing Parent with least cost

(Credits: Devin Connell & Hung Manh La)

Now the next algorithm => Rewire

```
Algorithm 3: $T$ ←Rewire($T$, $Q_{near}$, $q_{min}$, $q_{rand}$)
1 for $q_{near} \in Q_{near}$ do
2     $q_{path}$ ←Steer($q_{rand}$, $q_{near}$)
3     if ObstacleFree($q_{path}$) and Cost($q_{rand}$) + c($q_{path}$) <
        Cost($q_{near}$) then
4         $T$ ←ReConnect($q_{rand}$, $q_{near}$, $T$)
5     end
6 end
7 return $T$
```

Figure 5: Pseudo Code 3 – Rewiring the tree to get least cost back to the start node

(Credits: Devin Connell & Hung Manh La)

Similar to the ChooseParent algorithm, this too iterates through each member of the nearest neighbors list. Generate the new path. Next two checks are done a.) if the generates path isn't in obstacle space b.) sum of cost of the path and the new node is less than the cost of the nearest node. If and only if both of them are True, the we reconnect the new node the path with the least cost back to the start node.

C. *Difference between simple RRT and RRT**

The difference between this implementation and the general RRT start is that in the RRT algorithm the once the new node is generated, the path is checked for collision, if not then it is added as a path and a new random node is generated. It neither checks for a parent with least cost back to start nor does it rewire the tree based on that. Which is why RRT is computationally quicker but doesn't necessarily generate the most optimum path from the start to the goal node. Whereas RRT start has a good balance between speed and cost.

IV. IMPLEMENTATION

Our implementation of the project is using the RRT* algorithm, creating custom obstacle space, generate path and visualize the RRT* tree using Pygame visualization tool.

A. *First we designed the custom world in pygame*

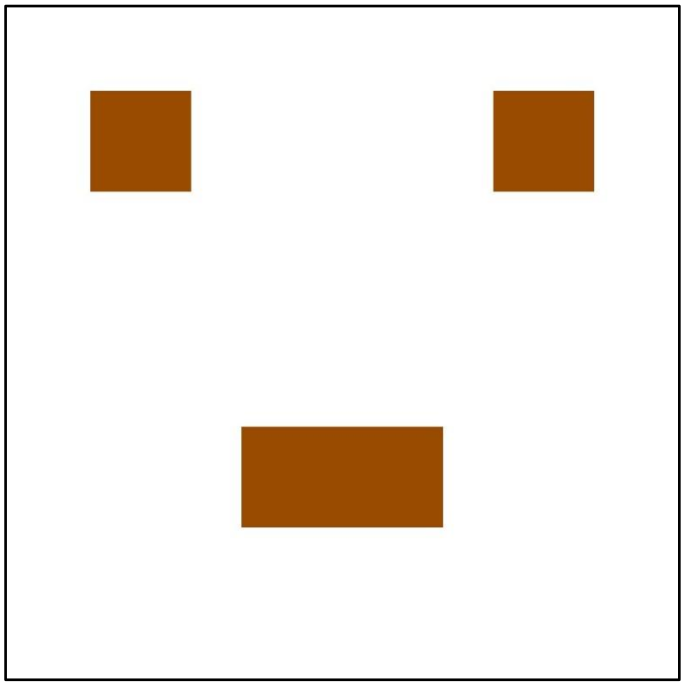

Figure 6: Obstacle space of the map

B. *Code organization*

The code has been written in Python. The visualization for the path is done using Pygame. Pygame has been used since the obstacle creation is pretty simple as compared to using Matplotlib.

At file level the code is split into two files.

a.) the first file contains the basic runner code. It is called as RRT_Star_*Main.py*

b.) The second is *Functions.py* file which contains all the necessary functions to run the code.

We will only run the RRT_Star_Main.py file since it will import all the items from the Function.py file.

C. *Outputs*

Once executed the program will provide with the following outputs

In figure 7, we can see that for the provided start and goal points a path had been generated by the algorithm. The explored nodes are shown in black and the path is shown in crimson red.

In figure 8, you can see how the rewiring is done. The old path has been shown in red. But since the tree has been rewired, it is no longer continued, so you can see that the branch ends there and rewiring is done.

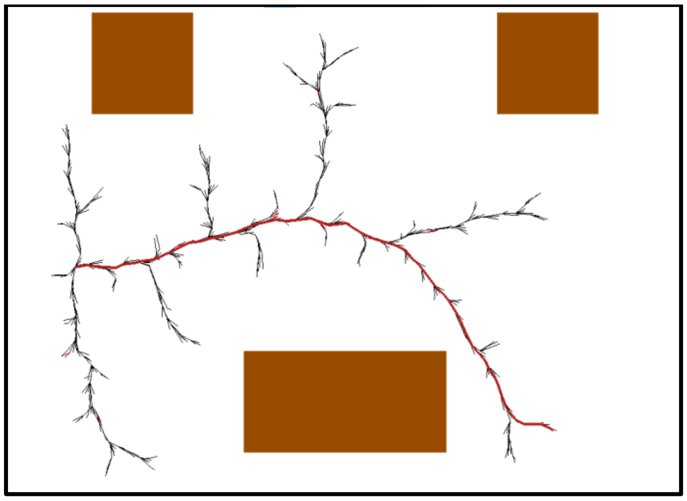

Figure 7: Output 1 of the RRT* algorithm

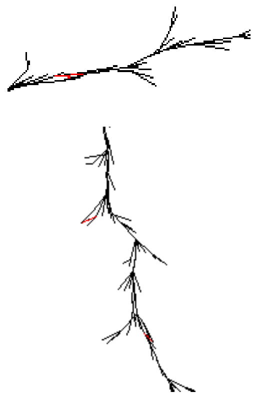

Figure 8: Rewiring in action

As we can see the rewiring is taking place properly. Below in figure 9 you can see the algorithm working for a different start and goal points.

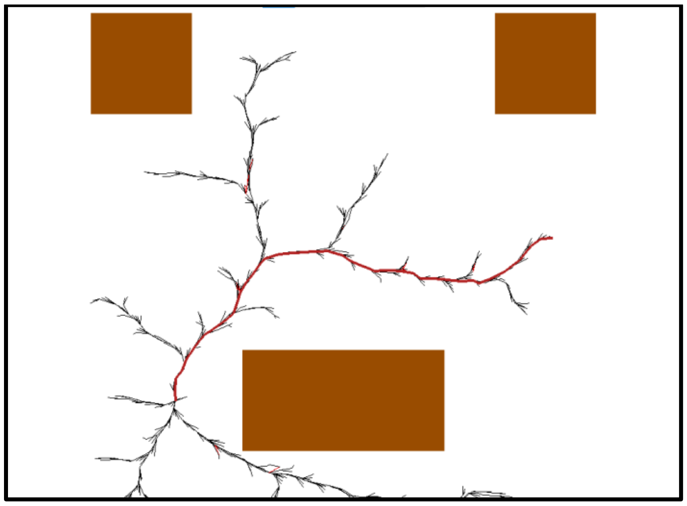

Figure 9: Output 2 of the RRT* algorithm

V. Learnings

In this project, we learnt about different types of sampling-based path planning methods. We implemented the RRT* algorithm and successively got the output for the obstacle space defined by us.

VI. goals accomplished

Of all the goals we set for ourselves in the proposal phase we were able to achieve the basic goal of implementing the RRT* algorithm. We weren't able to translate the results obtained into simulation in Gazebo using ROS and Turtlebot3. To offer ourselves a fallback plan, we proposed to deliver the 2-D visualization using either Matplotlib or Pygame which we have done successfully.

VII. Future improvements

In the future, we can apply non-holonomic constrains (even though the plain RRT* algorithm does give a non-holonomic constrains friendly output), get the x and y positions along the map. Use odometry tools in ROS and move the turtlebot using a closed loop controller.

References

[1] Paul Bourgine, Francisco J. Varela, "Towards a Practice of Autonomous Systems," Proceeding of the First European Conference on Artificial Life, Paris, France, 1991.

[2] Talal Husseini, "From Cherno-bots to Iron Man suits: the development of nuclear waste robotics," Power Technology, December 2018 - *updated in February 2022.*

[3] Prof. Reza Monferadi, "Plannig for Autonomous Robots," University of Maryland – College Park, Class Session Slides, January 2022 – May 2022.

[4] Prof. Pieter Abbeel , "Sampling-Based Motion Planning," University of California – Berkely, Fall 2012.

[5] Canberk Suat Gurel, Rajendra Mayavan Rajendran Sathyam & Akash Guha, "ROS-based Path Planning for Turtlebot Robot using Rapidly Exploring Random Trees (RRT*)," May 2018.

[6] L. Chang-an, C. Jin-gang, L. Guo-dong, and L. Chun-yang, "Mobile Robot Path Planning Based on an Improved Rapidly-exploring Random Tree in Unknown Environment," IEEE International Conference on Automation and Logistics, pp. 2375-2379, September 2008.

[7] C. Moon and W. Chung, "Kinodynamic Planner Dual-Tree RRT (DT-RRT) for Two-wheeled Mobile Robots using the Rapidly Exploring Random Tree," IEEE Transactions On Industrial Electronics, vol. 62, no. 2, pp. 1080-1090, February 2015.

[8] LaValle, Steven M., "Rapidly-exploring random trees: A new tool for path planning,". Technical Report. Computer Science Department, Iowa State University (TR 98–11), October 1998.

[9] LaValle, Steven M.; Kuffner Jr., James J., "Randomized Kinodynamic Planning," The International Journal of Robotics Research (IJRR), 20 (5): 378–400, 2001

[10] Karaman, Sertac, and Emilio Frazzoli. "Sampling-Based Algorithms for Optimal Motion Planning." The International Journal of Robotics Research, vol. 30, no. 7, June 2011, pp. 846–894.

[11] LaValle, S., "Planning Algorithms," Cambridge: Cambridge University Press, 2006.

[12] Devin Connel, Hung Manh La., "Dynamic Path Planning and Replanning with RRT*," Cornell University, April 2017.