University of Kentucky
UKnowledge

Theses and Dissertations--Mechanical Engineering

Mechanical Engineering

2023

Parallel Real Time RRT*: An RRT* Based Path Planning Process

David Yackzan
University of Kentucky, dwyackzan@gmail.com
Author ORCID Identifier:
https://orcid.org/0000-0002-8650-5063
Digital Object Identifier: https://doi.org/10.13023/etd.2023.194

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Yackzan, David, "Parallel Real Time RRT*: An RRT* Based Path Planning Process" (2023). *Theses and Dissertations--Mechanical Engineering*. 211.
https://uknowledge.uky.edu/me_etds/211

This Master's Thesis is brought to you for free and open access by the Mechanical Engineering at UKnowledge. It has been accepted for inclusion in Theses and Dissertations--Mechanical Engineering by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

David Yackzan, Student

Dr. Hasan Poonawala, Major Professor

Dr. Jonathan Wenk, Director of Graduate Studies

PARALLEL REAL TIME RRT*: AN RRT* BASED PATH PLANNING PROCESS

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in the College of Engineering
at the University of Kentucky

By

David Yackzan

Lexington, Kentucky

Director: Dr. Hasan Poonawala, Professor of Mechanical Engineering

Lexington, Kentucky

2023

Copyright ©David Yackzan 2023
https://orcid.org/0000-0002-8650-5063

ABSTRACT OF THESIS

PARALLEL REAL TIME RRT*: AN RRT* BASED PATH PLANNING PROCESS

This thesis presents a new parallelized real-time path planning process. This process is an extension of the Real-Time Rapidly Exploring Random Trees* (RT-RRT*) algorithm developed by Naderi et al in 2015 [1]. The RT-RRT* algorithm was demonstrated on a simulated two-dimensional dynamic environment while finding paths to a varying target state. We demonstrate that the original algorithm is incapable of running at a sufficient rate for control of a 7-degree-of-freedom (7-DoF) robotic arm while maintaining a path planning tree in 7 dimensions. This limitation is due to the complexity of maintaining a tree in a high-dimensional space and the network frequency requirements of the control signal for a real robotic system.

We develop and implement a parallelized version of RT-RRT*, dubbed Parallel RT-RRT* (PRT-RRT*), that can update motion plans in a dynamic environment while sending control signals at a high frequency. To achieve this, PRT-RRT* establishes a method of efficient communication between separate collision detection, path planning, and control nodes. We show that PRT-RRT* is capable of solving the dynamic path-planning problem on the 7D Franka Emika Panda robotic arm.

KEYWORDS: Path-Planning, Robotics, Dynamic Environments

David Yackzan

04/28/2023

Date

PARALLEL REAL TIME RRT*: AN RRT* BASED PATH PLANNING PROCESS

By
David Yackzan

Hasan Poonawala
Director of Thesis

Jonathan Wenk
Director of Graduate Studies

04/28/2023
Date

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. Poonawala for consistently providing guidance and support through the graduate school process. His constant willingness to provide a space to work through ideas and challenges has been the catalyst of much of my learning. He has supported me as an advisor, teacher, and mentor since I became a part of the lab.

Second, I would like to thank my family for their love and support in all aspects of my life. I would not be able to devote so much of my time and energy toward my goals without the unwavering support they have provided me.

Third, I would like to thank my colleagues, Benton Clark and Pouya Samanipour for always providing feedback and moral support throughout our shared graduate school journey.

Finally, I would like to thank Dr. Seigler and Dr. Hoagg for serving on my committee.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
LIST OF TABLES . . . vi
LIST OF FIGURES . . . vii
1 INTRODUCTION AND CONTRIBUTIONS . . . 1
1.1 Contributions . . . 2
2 LITERATURE REVIEW . . . 3
2.1 Reinforcement Learning (RL) . . . 3
2.2 Potential Fields (PFs) . . . 3
2.3 Control Barrier Functions (CBFs) . . . 4
2.4 Search Methods . . . 5
2.5 Real Time Search Methods . . . 5
2.6 Path Planning with Dynamic Obstacles . . . 5
2.7 Real-Time Motion Planning . . . 6
2.8 Fast Marching Trees . . . 6
3 BACKGROUND . . . 8
3.1 Problem Definition . . . 8
3.1.1 Static Path Planning Problem . . . 8
3.1.2 Dynamic Path Planning Problem . . . 8
3.1.3 Path Planning Tree . . . 9
3.2 Existing Algorithms . . . 11
3.2.1 RRT . . . 11
3.2.2 RRT* . . . 13
3.3 RT-RRT* . . . 15
3.4 PRT-RRT* Software Design . . . 20
3.5 PRT-RRT* Component . . . 20
3.5.1 Planner Component . . . 21
3.5.2 Controller Component . . . 25
3.5.3 Collision Checker Component . . . 25
3.5.4 Sensor Component . . . 26
4 SIMULATION METHODS AND RESULTS . . . 28
4.1 Evaluation of RT-RRT* Real Time Performance for a 7D Robotic System 28
4.2 Evaluation of Root Rewiring Impact on Executed Path Cost . . . 30
4.3 Evaluation of PRT-RRT* Tree Maintenance vs. Planning From Scratch 33
4.3.1 Comparative Scenarios . . . 33
5 DISCUSSION . . . 38
5.1 Value of Parallezing RT-RRT* . . . 38
5.2 Value of Preserving and Maintaining a Path Planning Tree . . . 39
5.3 Limitations . . . 40
5.3.1 Current Edge Obstructions . . . 40
5.4 Future Work . . . 40
5.4.1 Add Ability to React to Current Edge Obstructions . . . 40

5.4.2 Further Parallelization 41
APPENDICES 42
Appendix A 42
Appendix B 44
BIBLIOGRAPHY 45
VITA 50

LIST OF TABLES

4.1 RT-RRT* Single Node Rewiring Time 29
4.2 PRT-RRT* With vs. Without Root Rewiring Path Cost Results . . . 32
4.3 PRT-RRT* vs. M-RRT* Scenario Results 36

1 Panda PID Controller Gains 42
2 Simulation Hyperparameters 42
3 Root Rewiring Evaluation Independent T-Test Results 42
4 Tree Maintenance vs. Planning From Scratch Independent T-Test Results 43

LIST OF FIGURES

3.1 Example of a Path Planning Tree. 10
3.2 RT-RRT* Planner Process 17
3.3 RT-RRT* Root Advancement 18
3.4 RT-RRT* Root Advancement and Rewiring 19
3.5 PRT-RRT* Component Software Dependencies 21
3.6 PRT-RRT* Communication Between Components 22
3.7 PRT-RRT* Planner States 23
3.8 Example of Successful PRT-RRT* Reroute. 24
3.9 PRT-RRT* Controller States 26
3.10 PRT-RRT* Collision Checker States 27

4.1 M-RRT* Planner Process 31
4.2 Root Rewiring Evaluation States 32
4.3 Evaluation of Root Rewiring Impact on Executed Path Cost Plot . . 33
4.4 No Obstacle Scenario States 34
4.5 Add Ball Scenario States 35
4.6 Add Wall Scenario States 35
4.7 Scenario Cost Results Plot 37
4.8 Scenario Iteration Results Plot 37

CHAPTER 1. INTRODUCTION AND CONTRIBUTIONS

This thesis presents a new real-time path planning process called Parallel Real-Time RRT* (PRT-RRT*) along with an implementation of the algorithm for a 7-DoF robot arm in a dynamic environment. The goal of this work is to create a general-purpose, open-source method for quickly finding and executing collision-free paths from an agent to a variable target state in an environment with moving obstacles. By general-purpose, we mean this method should be easy to implement on a new robot in a new environment without requiring expertise. The implementation presented in this thesis used a Franka Emika Panda robotic arm with a 7-dimensional joint configuration space. The PRT-RRT* process is based on the Real Time RRT* (RT-RRT*) algorithm introduced in [1].

The RT-RRT* algorithm contributed strategies to the standard RRT* algorithm to advance and rewire around the planning tree root node online. These strategies allow a planning tree to be maintained and updated as the agent state, target state, and obstacle states vary. However, the root rewiring strategy becomes increasingly computationally expensive as the dimensionality of the path planning space increases. In 7 dimensions, it is infeasible to complete a significant amount of root rewiring in a loop that runs at a frequency high enough to execute control in serial. The authors demonstrate the capability of RT-RRT* to find and execute solution paths in a 2-dimensional configuration space in [1]. However, in Section 4.1 we show the shortcomings of that algorithm applied to a 7-dimensional configuration space in Section 4.1.

The PRT-RRT* algorithm distributes the computational expense of the planning, dynamic collision checking, and control sub-processes by running them on separate, parallel threads. This parallelization allows for the algorithm to find and execute dynamic path planning solutions for the 7-Dimensional state space of the Franka Emika

Panda robotic arm while maintaining a stored planning tree and sending control command signals at high frequency.

1.1 Contributions

We contribute the following:

- PRT-RRT*: A parallelization of RT-RRT* that enables the tree maintenance processes developed in RT-RRT* to be able to run on the 7D Panda robotic arm while sending control signals at the required frequency. Standard RT-RRT* is not able to run quickly enough to maintain the path planning tree and send control signals in the Panda robot arm configuration space.

- An easily transferable software package to implement the PRT-RRT* planning process an any robot with a URDF and a simple controller.

The software package with documentation for implementing the PRT-RRT* process on a robot within the ROS framework can be found at [2].

CHAPTER 2. LITERATURE REVIEW

In this section we discuss various methods in the literature that explore solutions to navigating robots in environments with obstacles.

2.1 Reinforcement Learning (RL)

Many RL strategies have been successfully employed for collision-free navigation of robots in dynamic environments. One example presented in [3] demonstrates the successful control of a robot manipulator around a moving human in simulation using a Deep RL policy. Another RL example dubbed Mobile robot Collision Avoidance Learning (MCAL) [4], demonstrates the ability to control a wheeled SR7 robot in an environment with reciprocating obstacles using a Soft Actor Critic policy. The work in [4] also incorporates path planning in a second method dubbed Mobile robot Collision Avoidance Learning with Path (MCAL_P) which uses the trained RL policy to follow planned solution paths.

The disadvantage of RL approaches is that they do not provide easily transferable solutions. If a new obstacle is added into an environment, or if the robot changes, then the RL algorithm will require retraining before it can be re-implemented. This training process requires expertise and time to conduct properly.

2.2 Potential Fields (PFs)

PF-based path planning methods [5] overlay the configuration space of a robot with a vector field that repulses the robot state from obstacles and attracts the robot state towards the target state. The vector field derives from the gradient of a potential function chosen such that the target state is a minimum of it. These methods are capable of tracking variable target states in environments with moving obstacles, however, there are several limitations. Often, the target state is not the only minimum. The other minima represent trapped states from which the agent will not be able to leave and will never reach the target state. For example, if the

agent reaches a state in the vector field that equally repels the agent from advancing due to nearby obstacles and equally attracts the agent to the target state, then the agent can get stuck. PFs can also produce cyclic motion in which the agent moves back and forth without achieving the target state due to repelling obstacles. PF-based navigation methods are also unable to pass between closely spaced obstacles and struggle with pick-and-place applications using robot arms as the target state is necessarily near an obstacle. These limitations are explored in detail in [6].

2.3 Control Barrier Functions (CBFs)

CBFs have been successfully developed for safely achieving desired target states while avoiding static obstacles and even other controllable agents in an environment. The primary advantage of CBFs is that they can be used to provide safety guarantees when the dynamics of an environment are known and controllable. A recent example of a successful CBF application is presented in [7] where the authors use probabilistic movement primitive distributions to define Control Lyapunov Functions (CLFs) and CBFs. The authors report their method is capable of controlling a system around static obstacles without leaving a neighborhood defined by a training set. They demonstrate their method by creating a controller for a UR5e robot in simulation that produces collision-free trajectories in an environment with obstacles. The work presented in [8] shows a collision-free supervisory control of several agents in an environment with limited actuation. The authors of the paper present obstacle avoidance guarantees and demonstrate the methods in a simulation with drones.

CBF solutions are low-level solutions to obstacle avoidance navigation tailored to the dynamics of the systems and environments they are designed for. They can be implemented in conjunction with higher level path-planning based solutions to combine long-term path benefits of planning with short-term safety benefits of CBFs. A solution using planning and CBFs can be seen in [9] in which the Control Barrier Function guided Rapidly-exploring Random Trees (CBF-RRT) algorithm is presented.

2.4 Search Methods

A* search [10] is a well-known search algorithm capable of finding the optimal path between two nodes in a graph. However A* search does not scale well, and sub-optimal versions such as weighted A* search [11] were produced that trade efficiency for solution quality. The work in [12] presents a novel search method, R* search, that focuses on batched, easy-to-solve searches for motion planning. This method allows the algorithm to avoid local minima and also discard search state-space from memory once it completes each of its search batches, allowing it to scale well. The authors show experimentally that R* search is able to scale to large complex planning problems.

2.5 Real Time Search Methods

Real time path planning search methods were developed based on A* and R* search, called Real-time A* (RTA*) [13] and Real-time R* (RTR*) [14] respectively. The authors of RTR* demonstrate its ability to plan paths online for an agent to avoid a moving obstacle while seeking a goal state in a 2D simulation in [14].

2.6 Path Planning with Dynamic Obstacles

In other related work, algorithms have been developed to work around moving obstacles by considering their dynamics while planning paths. In [15], the authors introduce a method by which a car can be automatically navigated around other moving cars by tracking and predicting their trajectories. In [16] the authors add a "safe interval" variable to states in the path planning tree that encodes the amount of time-steps in the future that the state will be considered collision free. This enables paths to be chosen safely, where a solution path is only considered if the safe interval ensures the states will be collision free long enough for the agent to move through them. The work in [17] considers the case where multiple agents in an environment react to each other. In this work the authors demonstrate the concept of optimal reciprocal collision avoidance in which the potential reactive motion of other agents in

the environment is accounted for. The drawback of these methods is that they assume the obstacle dynamics and reciprocal motion of other agents in the environment are known.

2.7 Real-Time Motion Planning

In [18], the authors introduce parallelization techniques for common path planning operations such as random sampling and nearest neighbor computation using Graphics Processing Units (GPUs) which they call "g-Planner". In [19] authors from the same lab introduce a highly efficient parallelization technique for collision checking within the path planning framework. The parallelization of these techniques allow for much faster real-time motion planning. Similar work is presented in [20] in which the authors present work to construct field-programmable gate array (FPGA) chips specifically for the task of motion planning. The authors state that these chips are capable of performing motion planning calculations three orders of magnitude faster than existing methods. While these real-time motion planning techniques are capable of exponentially improving the efficiency of path planning processes, the main drawback of this work is that it requires users to purchase and install specific hardware.

The work in [21] demonstrates another application of real-time motion planning using a sparse Probabilistic Roadmap (PRM) with a smoothing neural network (NN) to quickly solve for a smooth path from one point to another. The authors successfully implement their work on a 6 DoF robotic arm in a simulation that is able to quickly react to obstructing obstacles. The drawback of this work is that in order to implement it on a different system, the smoothing NN must first be trained, which requires expertise and time.

2.8 Fast Marching Trees

The Fast Marching Trees (FMT*) algorithm, introduced in [22] is a sampling-based motion planning algorithm that is capable of solving complex problems in high-dimensional configuration spaces. FMT* is an extension of Fast Marching Methods

(FMM) which were introduced in discussed in [23],[24] as numerical methods for calculating solutions to the Eikonal equation. Solutions to the Eikonal equation can be used for computing shortest paths in continuous domains. More recent work, introduced in [25] extends the FMT* method to work efficiently for collision-free path planning around obstacles. They dub the extended algorithm obstacle-based fast marching tree (OB-FMT*). This work is similar to other recent work introduced in [26], in which the authors introduce an obstacle-aware sampling method that ensures new samples from the configuration space will not be obstructed by obstacles. They show that their sampling method significantly lowers the time to solve and the path length when implemented with RRT as opposed to RRT with non-obstacle-aware sampling methods.

CHAPTER 3. BACKGROUND

In this section, we first formulate the static and dynamic path planning problem and provide definitions for the tree data structure in the scope of this work. We move on to discuss the RRT, RRT*, and RT-RRT* algorithms that PRT-RRT* is built on.

3.1 Problem Definition

3.1.1 Static Path Planning Problem

The static path planning problem solved by RRT and its variants is defined over the configuration space of an agent. The dimension of the configuration space is equivalent to the degrees of freedom of the agent and will be denoted by n. Thus, the configuration space is defined as $Q \subset \mathbb{R}^n$. The Franka Emika Panda robot arm the simulations are run on in this work has a dimension of $n = 7$, and thus its configuration space is defined as $Q \subset \mathbb{R}^7$. The portion of the configuration space obstructed by obstacles will be denoted by $Q_{obs} \subset Q$ and the portion of the state space unobstructed by obstacles will be denoted by $Q_{free} \subset Q$. Note that $Q_{free} = Q \setminus Q_{obs}$.

A static path planning problem is initialized with the following data: the configuration state space Q, the current state of the agent $q_{curr} \in Q$, the target state $q_{target} \in Q$, and the portion of the state space obstructed by obstacles Q_{obs}. The static path planning problem is valid as long as $q_{curr} \in Q_{free}$, $q_{target} \in Q_{free}$, and there is some path, defined as $E_{target} \in Q_{free}$ that exists between them. The static path planning problem is considered solved when E_{target} is found. A survey of 8 prominent static path planning methods for mobile robots can be found in [27].

3.1.2 Dynamic Path Planning Problem

The dynamic path planning problem is a variation of the static path planning problem defined above in Section 3.1.1. The dynamic path planning problem is also defined over the configuration state space of an agent $Q \subset \mathbb{R}^n$. The main difference

being that the dynamic path planning problem varies over time t.

In the dynamic path planning problem the agent state $q(t)_{curr} \in Q$, the target state $q(t)_{target} \in Q$, and the space obstructed by obstacles $Q(t)_{obs} \subset Q$ can change throughout the problem as a function of time. We define the space obstructed by dynamic obstacles at time t as $Q(t)_{obs,dyn} \in Q$, and define the space obstructed by static obstacles as $Q_{obs,st} \in Q$. The obstacle space $Q(t)_{obs}$ is now defined as the space obstructed by both static and dynamic obstacles such that $Q(t)_{obs} = Q(t)_{obs,dyn} \cup Q_{obs,st}$. The free space also becomes a function of time, $Q(t)_{free} \subset Q$ and $Q(t)_{free} = Q \setminus Q(t)_{obs}$ still holds.

The dynamic path planning problem is valid as long as $q(0)_{curr} \in Q(0)_{free}$, $q(t)_{target} \in Q(t)_{free}$ $\forall t$, and there is some path, defined as $E(t)_{target} \in Q(t)_{free}$ that exists between them at all times t. The dynamic path planning problem is considered solved when the current state of the agent $q(t)_{curr}$ has reached the target state $q(t)_{target}$ at time t such that $q(t)_{curr} = q(t)_{target}$.

3.1.3 Path Planning Tree

Upon initialization of the path planning problem, an RRT based solution algorithm creates a tree, defined by $\mathcal{T} \subset Q$. The tree is made up of a collection of nodes, denoted by $q_i \in Q$, and the edges between them. An edge is defined as the line directly connecting two nodes, denoted by $e_{i,j} \subset Q$ where $e_{i,j}$ is the edge between q_i and q_j. If a node is in the tree, we say $q_i \in \mathcal{T}$. If an edge is in the tree, we say $e_{i,j} \in \mathcal{T}_e$. An example of a path planning tree is illustrated in Figure 3.1.

Note that while some of the following terms will be a function of time t in the dynamic path planning problem, we omit t for simplification. A path planning tree is initialized with the root node $q_{root} \in Q$ such that $q_{root} = q_{curr}$. Each node $q_i \in \mathcal{T}$ has one or zero parent node(s) $p_i \in Q, \mathcal{T}$ (zero only in the case of the root node), zero or more child nodes $h_i \subset Q, \mathcal{T}$, and an associated cost $c_i \in \mathbb{R}$. An edge exists in the

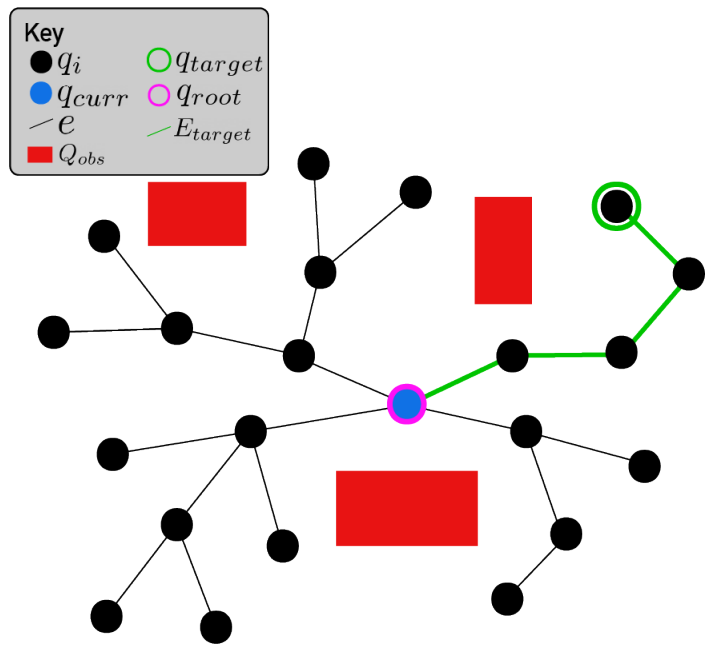


Figure 3.1: Example of a Path Planning Tree. The dots represent states in the tree, the green circle q_{target} encircles the target state, the blue dot q_{curr} represents the current state of the agent, the pink circle q_{root} encircles the current root of the path planning tree, the black lines e represent edges in the tree, the green lines E_{target} represent a solution path to the target state, and the red squares Q_{obs} represent obstacles in the environment.

tree $e_{i,j} \in \mathcal{T}_e$ between two nodes q_i, q_j if either q_i is the parent of q_j ($q_i = p_j$) or q_i is a child of q_j ($q_i \in h_j$). A node with zero child nodes is called a leaf node.

The cost between any two nodes in the tree $q_i, q_j \in \mathcal{T}$, denoted by $c_{i,j}$ or $c_{j,i}$ is computed by a norm distance function over the edge between the two nodes. We use the 1-norm in the implementation developed in this thesis, thus

$$c_{i,j} = \|e_{i,j}\|_1 = \|q_i - q_j\|_1 = |q_{i,1} - q_{j,1}| + |q_{i,2} - q_{j,2}| + ... + |q_{i,n} + q_{j,n}|. \tag{3.1}$$

A path, denoted by E_i is defined as the set of concurrent edges in the tree that lead from any node in the tree $q_i \in \mathcal{T}$ back to the root node of the tree $q_{root} \in \mathcal{T}$. Note that since every node other than q_{root} in the tree has exactly one parent node, it follows that there is exactly one path for each node. The path for the root node is

the empty set $E_{root} = \emptyset$. The cost c_i associated with a node q_i is computed by the sum of all the edges in its path E_i. Thus, the cost of a node q_i is computed by

$$c_i = \sum_{e \in E_i} \|e\|_1. \tag{3.2}$$

Each path E_i has an associated list of nodes, which will be defined by $Q_i \subset Q$. For example, let E_{10} be the path to the 10th node added to the planning tree such that $E_{10} = \{e_{root,4},\ e_{4,7},\ e_{7,10}\}$. Then the associated list of nodes to E_{10} would be $Q_{10} = \{q_{root}, q_4, q_7, q_{10}\}$. Note that Q_i will always start with q_{root} and end with q_i.

3.2 Existing Algorithms

3.2.1 RRT

The Rapidly exploring Random Trees (RRT) algorithm was developed in [28] as a solution to the static path planning problem. It is an iterative process that uses random sampling to quickly build a space-filling tree, capable of finding feasible solution paths in high-dimensional spaces efficiently. The algorithm is outlined in Algorithm 1

The static path planning problem is initialized as defined in Subsection 3.1.1 with Q, q_{curr}, q_{target}, Q_{obs}. The RRT algorithm begins by establishing a tree $\mathcal{T}$ with $q_{root} \in \mathcal{T}$ such that $q_{root} = q_{curr}$.

Algorithm 1: RRT

```
Input: q_root, q_target, Q_obs
T ← q_root
while q_target ∉ T do
    q_rand = Sample(Q)
    if q_rand ∉ Q_obs then
        q_near = FindNearestNode(T, q_rand)
        if Distance(q_near, q_rand) > d_max then
            q_new = Steer(q_near, q_rand)
        else
            q_new = q_rand
        end
        if e_near,new ∉ Q_obs then
            c_new = c_near + c_near,new
            T = q_new, e_near,new
        end
    end
end
Output: E_target
```

For each RRT iteration, a random point q_{rand} is sampled uniformly from the configuration space. If $q_{rand} \in Q_{obs}$, then a new random point is sampled until $q_{rand} \in Q_{free}$. The nearest node in the tree to q_{rand} is found, denoted by $q_{near} \in \mathcal{T}$. If the sampled point q_{rand} is within a maximum distance d_{max} of q_{near}, then a new node is defined by $q_{near} = q_{rand}$. Otherwise, a steering function is used to interpolate a node closer to q_{near} as $q_{new} = \text{Steer}(q_{rand}, q_{near})$. A common steering function and the one employed in the implementation developed in this thesis is defined by

$$\text{Steer}(q_{rand}, q_{near}) = d_{max} \left(\frac{q_{rand} - q_{near}}{||q_{rand} - q_{near}||} \right).$$

If the edge between q_{near} and q_{new} is not in collision with any obstacles such that $e_{near,new} \notin Q_{obs}$, then q_{new} is added to the tree as a child to q_{near}. These steps are repeated until a new node sufficiently close to q_{target} is added to the tree.

It may take many iterations for an RRT algorithm to find a solution path to a target state using random sampling. To mitigate this, approaches have been developed as

discussed in [29] to bias the tree growth and guide the search. Define a random number $x \in [0, 1]$, and let α be a small, pre-defined hyperparameter. For this work, we use the goal-biased sampling method with $\alpha = 0.05$ as follows

$$q_{rand} = \begin{cases} q_{target}, & \text{if } x < \alpha \\ \text{RandomSample}(Q), & \text{otherwise} \end{cases} \tag{3.3}$$

3.2.2 RRT*

The RRT* algorithm developed in [30] improved RRT with the addition of two key strategies: a method to choose the best parent when adding a new node to the tree and a rewiring method to check for path improvements through a newly added node to the tree. These strategies decrease the cost of the solution path as iterations proceed. The RRT* algorithm is outlined in Algorithm 2 with the two key strategies highlighted.

Algorithm 2: RRT*

Input: $q_{root}, q_{target}, Q_{obs}$
$\mathcal{T} \leftarrow q_{root}$
while $q_{target} \notin \mathcal{T}$ **do**
  $q_{rand} = \text{Sample}(Q)$
  if $q_{rand} \notin Q_{obs}$ **then**
    $q_{near} = \text{FindNearestNode}(\mathcal{T}, q_{rand})$
    if $\text{Distance}(q_{near}, q_{rand}) > d_{max}$ **then**
      $q_{new} = \text{Steer}(q_{near}, q_{rand})$
    else
      $q_{new} = q_{rand}$
    end
    if $e_{near,new} \notin Q_{obs}$ **then**
      $Q_{nn} = \text{GetNeighbors}(\mathcal{T}, q_{new})$
      $\text{AddToBestParent}(\mathcal{T}, Q_{nn}, q_{near}, q_{new})$ using Algorithm 3
      $\text{RewireAroundNode}(\mathcal{T}, Q_{nn}, q_{new})$ using Algorithm 4
    end
  end
end
Output: E_{target}

Algorithm 3: Add To Best Parent

Input: $\mathcal{T}, Q_{nn}, q_{near}, q_{new}$
$c_{new} = c_{near} + c_{near,new}$
for $\underline{q_i \in Q_{nn}}$ **do**
  if $\underline{c_i + c_{i,new} < c_{new} \text{ and } e_{i,new} \notin Q_{obs}}$ **then**
    $c_{new} = c_i + c_{i,new}$
    $q_{near} = q_i$
  end
end
$\mathcal{T} \leftarrow q_{new}$, $\mathcal{T}_e \leftarrow e_{near,new}$

Algorithm 4: Rewire Around Node

Input: $\mathcal{T}, Q_{nn}, q_{new}$
for $\underline{q_i \in Q_{nn}}$ **do**
  if $\underline{c_{new} + c_{i,new} < c_i \text{ and } e_{i,new} \notin Q_{obs}}$ **then**
    $c_i = c_{new} + c_{i,new}$
    $\mathcal{T}_e \leftarrow e_{i,new}$
  end
end

The method to choose the best parent in the tree for a new node (Algorithm 3) is incorporated after the edge $e_{near,new}$ is determined to be collision-free. Before adding q_{new} to the tree, RRT* first gathers the set of near neighbors to q_{new} in the tree as $Q_{nn} = \{q_i \in \mathcal{T} : ||q_{new} - q_i|| < d_{max}\}$. The algorithm then iterates through all of the nodes in Q_{nn}, computing the cost c_{new} for the new node as a child of each $q_i \in Q_{nn}$ and checking if the edge $e_{i,new}$ is collision-free and could be a valid connection. The algorithm chooses the parent of q_{new} (p_{new}) as the lowest-cost node in Q_{nn} where there is a valid connection.

The rewiring strategy (Algorithm 4) takes place after a new node is added to the tree. At the end of an RRT* iteration where a new node q_{new} was added to the tree, the algorithm attempts to rewire any nodes in Q_{nn} through q_{new}. For each neighbor node $q_i \in Q_{nn}$ if the cost of the neighbor node c_i is lowered when the parent of the neighbor node p_i is changed to q_{new}, then the neighbor node will be removed from its previous parent and rewired in the tree with q_{new} as its parent node.

Improvements to RRT*. Many further improvements have been made to the RRT* algorithm since its development. Three improvements for RRT* are presented in [31] that allow RRT* to handle sporadic obstacles in the environment by pruning the tree, improve efficiency of solution path improvement by sampling around the first solution path found, and improve efficiency of rewiring by checking if a node can be directly connected to its grandparent rather than its parent. A method in which a planning tree was grown from both the target configuration and the start configuration at the same time called Bidirectional RRT* (B-RRT*) was introduced in [32]. And more recent improvements of the B-RRT* algorithm are introduced in [33],[34],[35],[36],[37]. Another method introduced in [38] called RRT-connect also attempts to grow two trees from the start and target configurations as in B-RRT*, but adds a step each iteration in which the trees attempt to connect to each other. More recently, an optimal version of RRT-connect called RRT*-connect was introduced in [39]. Even more recently, methods of combining machine learning with path planning have been explored by Jainkn Wang and others in [40],[41],[42].

3.3 RT-RRT*

The RT-RRT* algorithm developed in [1] presents an attempt to solve the dynamic path planning problem defined in Section 3.1.2. It builds on ideas developed in the RRT^X algorithm from [43],[44], in which the idea of real-time motion re-planning was introduced. The RT-RRT* algorithm is intended to execute a high frequency loop that maintains and updates a path planning tree while controlling an agent along collision-free paths to a target state. To attempt to keep the main loop running at a sufficiently high frequency to send control signals, the amount of time allowed for expansion and rewiring of the path planning tree, and control of the agent is restricted.

In contrast to RRT based static path planning problem solutions, RT-RRT* stores and maintains its path planning tree until the agent reaches the target state. As a

result the planning tree is maintained and expanded for a much longer time period in RT-RRT*, and the size of the tree must be limited to conserve memory. To this end, the authors incorporate a sampling density rejection algorithm such that a node will not be added to the tree if there are too many nodes in the space around it or it is too close to another node already in the tree. Let k_{max} be the maximum amount of nodes allowed in a neighborhood of a planning tree and let r_{min} be the minimum allowable distance between two nodes in the tree. The sampling density rejection algorithm is given in Algorithm 5.

Algorithm 5: Sampling Density Rejection

Input: $q_{new}, \mathcal{T}$
$q_{near} = \text{FindNearestNode}(\mathcal{T}, q_{new})$
$Q_{nn} = \text{GetNeighbors}(\mathcal{T}, q_{new})$
if $\underline{\text{Size}(Q_{nn}) < k_{max}}$ **or** $\underline{\|q_{near} - q_{new}\| \geq r_{min}}$ **then**
  $\text{AddToBestParent}(\mathcal{T}, Q_{nn}, q_{near}, q_{new})$ using Algorithm 3
end

The authors contribute two main tree maintenance methods to the base RRT* algorithm that keep the path planning tree up to date with the problem dynamics. One method advances the root of the tree as the state of the agent changes, and a second method rewires the tree around the updated root. Figure 3.2 illustrates the RT-RRT* planner process.

The root node advancement strategy advances the root node, q_{root} as the agent state q_{curr} is controlled along a solution path E_{target} such that the agent state is always moving towards the current root. This keeps the planning tree paths up to date as the agent moves in its state space. Note that the root node is advanced once the agent state q_{curr} is sufficiently close to q_{root}. For example, let $E_{target} = \{e_{root,4},\ e_{4,7},\ e_{7,10},\ e_{10,target}\}$, where the agent state q_{curr} is moving towards q_{root}. Once the agent state gets sufficiently close to q_{root} the root node will advance to $q_{root} = q_4$ and thus the solution path will be updated and shortened to $E_{target} = \{e_{root,7},\ e_{7,10},\ e_{10,target}\}$. This method is illustrated in Figure 3.3.

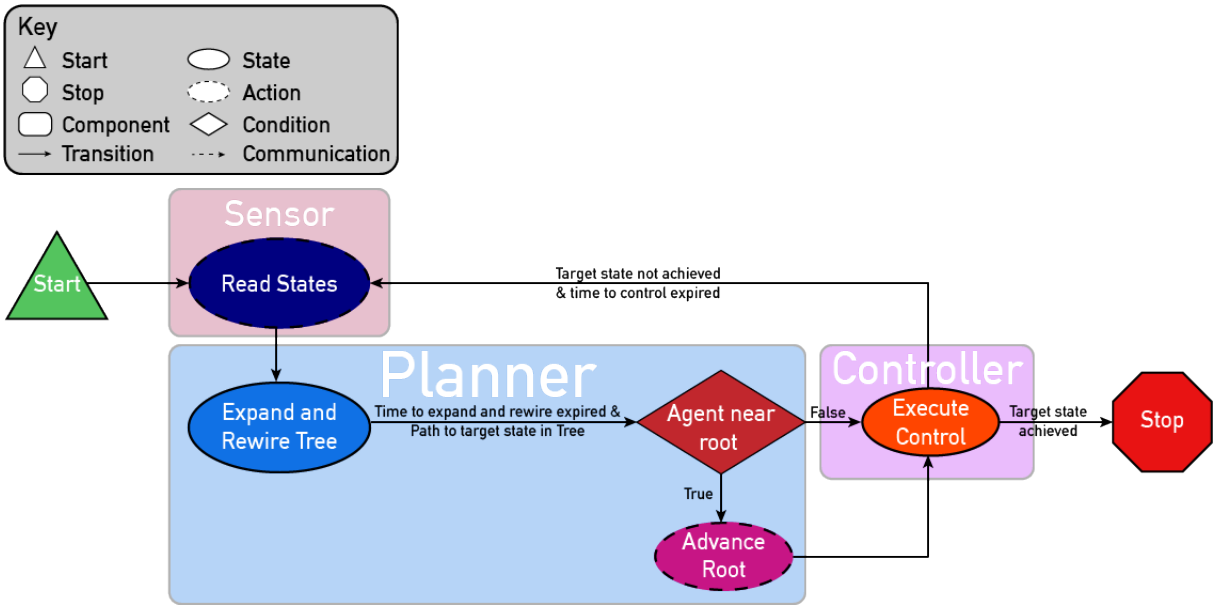
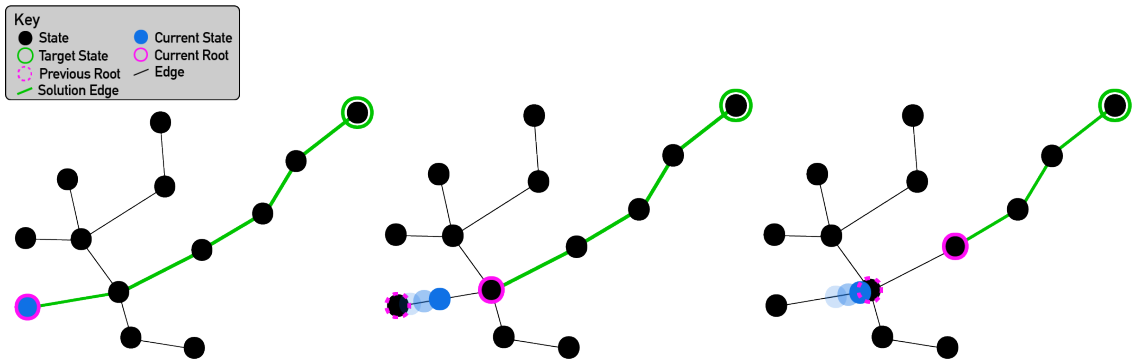


Figure 3.2: RT-RRT* Planner Process

The rewire around root strategy allows nodes to rewire along lower cost paths around the updated root node. The authors of RT-RRT* establish a root rewire priority queue, denoted by $Q_{root} \subset Q$, to hold nodes that will be rewired next. The highest priority nodes to be rewired next are at the front of the queue. To ensure resources are not wasted attempting to rewire a node more than once per iteration, a set $S_{root} \subset Q$ holds the nodes that have been rewired during the current rewiring iteration and is checked before adding new nodes into Q_{root}. The rewire around root algorithm is given in Algorithm 6.



(a) Solution path found (b) Root advanced to first state (c) Root advanced to second state

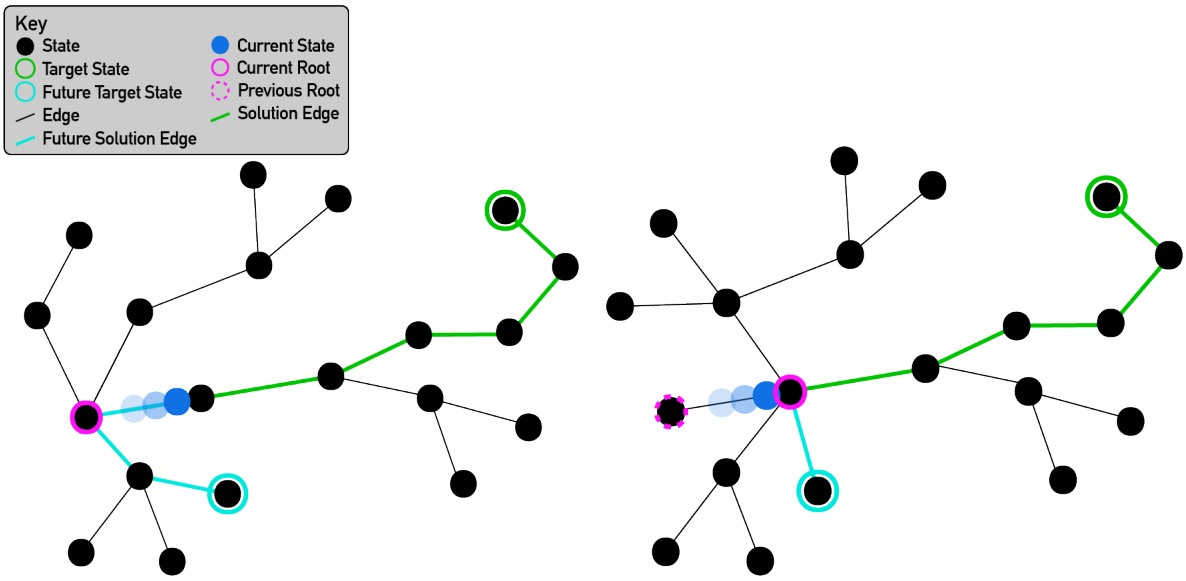
Figure 3.3: RT-RRT* Root Advancement. a) A solution path is found from the current state of the agent (blue dot) to the target state (green circle). b) The current root (pink circle) is advanced to the first state in the solution path, and the agent (blue dot) is controlled toward the current root. c) The agent (blue dot) approaches the first state in the solution path and the current root (pink circle) is advanced to the second state in the solution path.

Algorithm 6: Rewire Around Root

Input: $Q_{root}, \mathcal{T}$
$S_{root} = \emptyset$ `// To track which nodes get rewired`

if $Q_{root} = \emptyset$ **then**
  $Q_{root} \leftarrow q_{root}$
  $S_{root} \leftarrow q_{root}$
end
$\mathcal{T} \leftarrow q_{root}$
while Time not expired **and** $Q_{root} \neq \emptyset$ **do**
  $q_r = \text{PopFront}(Q_{root})$
  $Q_{nn} = \text{GetNeighbors}(\mathcal{T}, q_r)$
  for $q_i \in Q_{nn}$ **do**
    if $c_r + c_{i,r} < c_i$ **and** $e_{i,r} \notin Q_{obs}$ **then**
      $c_i = c_r + c_{i,new}$
      $\mathcal{T}_e \leftarrow e_{i,r}$
    end
    if $q_i \notin Q_{root,unique}$ **then**
      $Q_{root} \leftarrow q_i$
      $S_{root} \leftarrow q_{root}$
    end
  end
end

Without these methods, if the target state changes and is satisfied by a node on a different branch then the branch the current state of the agent is on, then a new solution path would necessarily travel through the original root location. This will likely not be the shortest solution path available through the nodes in the planning tree. This case is demonstrated in Figure 3.4. The combination of these strategies allow the planning tree to store the shortest possible paths along the nodes in the tree to the root node that best represents the agent's current state as it moves.



(a) Without root advancement and rewiring. (b) With root advancement and rewiring.

Figure 3.4: RT-RRT* Root Advancement and Rewiring. a) The current state of the agent (blue dot) is controlled along the initial solution path (green path) in a tree without tree without node advancement and rewiring. The path to the new target (cyan circle) will involve backtracking to the original root node (pink circle), traversing three edges (cyan path). b) The current state of the agent (blue dot) is controlled along the initial solution path (green path) in a tree maintained with root advancement and rewiring. The root node (pink circle) has changed to the node that the current state is approaching. Due to rewiring, the new target state is only one edge (cyan path) away from the node that the current state is approaching.

3.4 PRT-RRT* Software Design

We developed the PRT-RRT* implementation on the 7-DoF Franka Emika Panda robot arm using open-source software. The Robot Operating System (ROS) framework was utilized for managing communication between the planning, collision detection, control, and sensing processes. Each of these four processes will be referred to as a component. The controller component utilizes ROS MoveIt for high-level control within the ROS franka, libfranka for low-level control of the Panda robot, and Franka ROS to bridge the gap. The path planning process was implemented in the Open Motion Planning Library (OMPL) framework. The collision detector and the planner components both use the Bullet Continuous Collison Detection (CCD) library for collision detection. Figure 3.5 illustrates the software dependencies of each component.

The software package and documentation for installing and implementing PRT-RRT* for Panda or any other robot with a URDF can be found in [2]. Note that while the planner and collision detector components will work with other robots given a URDF, in order to implement this process for a new setup the sensor and controller processes will need to be designed for the robot and sensor stack in use. We use the ROS Joint Trajectory Controller interface for PID control of the Panda robot arm. A PID controller can be easily set up and tuned for a new robot as described in the documentation in [45]. The PID control gains for the Panda controller are reported in Table 1 in Appendix A. Note that any control may be used provided it is capable of reaching the neighborhood of goal configurations within a known finite time.

3.5 PRT-RRT* Component

There are four distinct components running in parallel that make up the PRT-RRT* process: the planner component, the controller component, the collision checker component, and the sensor component. See Figure 3.6 for a diagram of the communication between components.

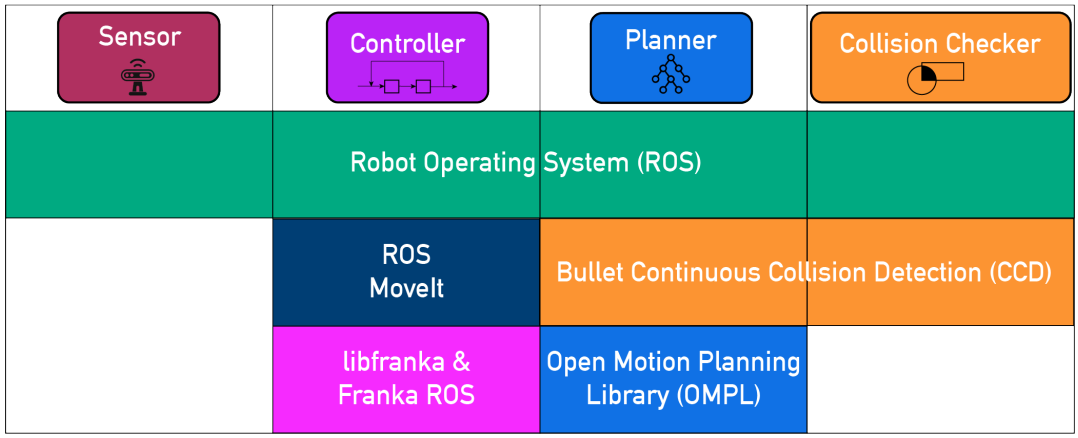


Figure 3.5: The PRT-RRT* components are laid out in the top row with the software packages used in the 3 rows below. The components were designed using the software packages within their individual columns.

3.5.1 Planner Component

The planner component is made up of four primary process states and the transitions between them. The states and state transitions are illustrated in Figure 3.7 and described below.

Tree Expansion. The planner component seeks solution paths to the planning problem, starting from the current state of the agent $q(t)_{curr}$ and ending at the current target state $q(t)_{target}$ $\forall t$. The planner process employs the RRT* algorithm discussed in Section 3.2.2 to expand and rewire the tree as it searches for solution paths. Once a solution path is found, the planner enters the communication state, publishes the path out to the other components, and waits to hear back.

Communication. This short waiting period is essential to keep the solution path information aligned between the parallel processes. For example, if the planner continues updating the tree and finds a better solution path, say $\hat{E}(t)_{target}$ after $E(t)_{target}$ was published, then the path to be executed by the controller and checked for collision by the collision checker will be misaligned with the path held by the planner ($\hat{E}(t)_{target} \neq E(t)_{target}$). Thus, the planner must wait to continue operating on the tree until it receives an update that the controller is executing the next step in the

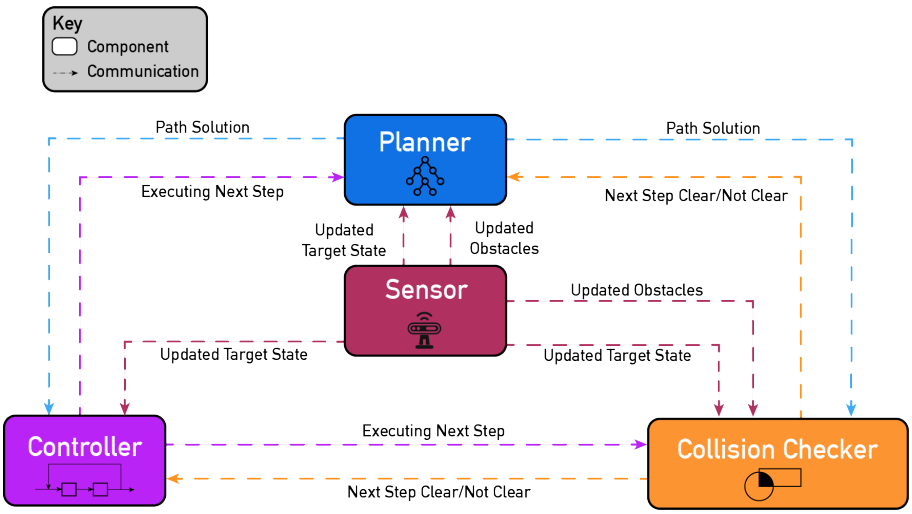


Figure 3.6: PRT-RRT* Communication Between Components

path or that the collision checker found the next step obstructed.

Tree Maintenance. Once the planner is notified that the control has begun executing to the next state in the solution path, the planner will transition to maintaining the planning tree. To maintain the planning tree, the planner component first advances the root node and then rewires from the root node similar to the RT-RRT* routines discussed in Section 3.2.2. Recall that in RT-RRT*, the root node is advanced when the agent state gets sufficiently close to the root node. In contrast, the PRT-RRT* planner does not advance the root node until the controller has started executing to a new state. The planner component will output an updated solution path if a better path is found during rewiring. The planner spends the majority of the maintenance period rewiring the tree, but also allocates some time to expand the tree using RRT* so that exploration of the space continues throughout the process. This exploration is useful in case the target state changes in the future.

Rerouting. If the next edge in the solution path becomes obstructed then the planner needs to seek an unobstructed path to the target. The collision checker component will alert the planner component to reroute. The reroute routine attempts to find a better parent for each of the nodes along the currently obstructed solution

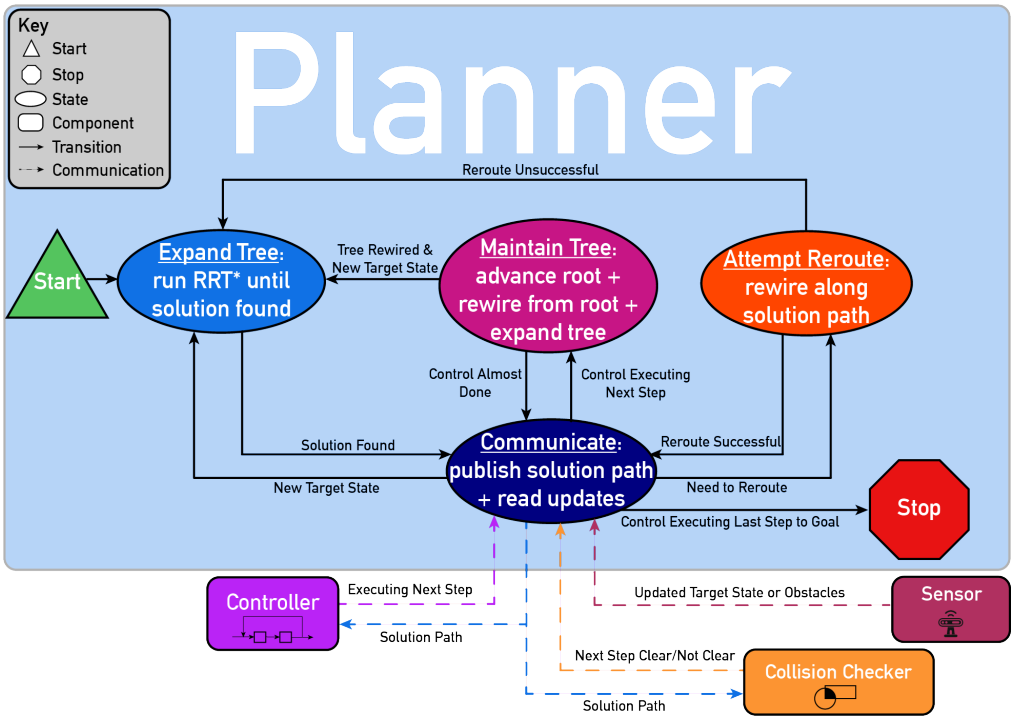


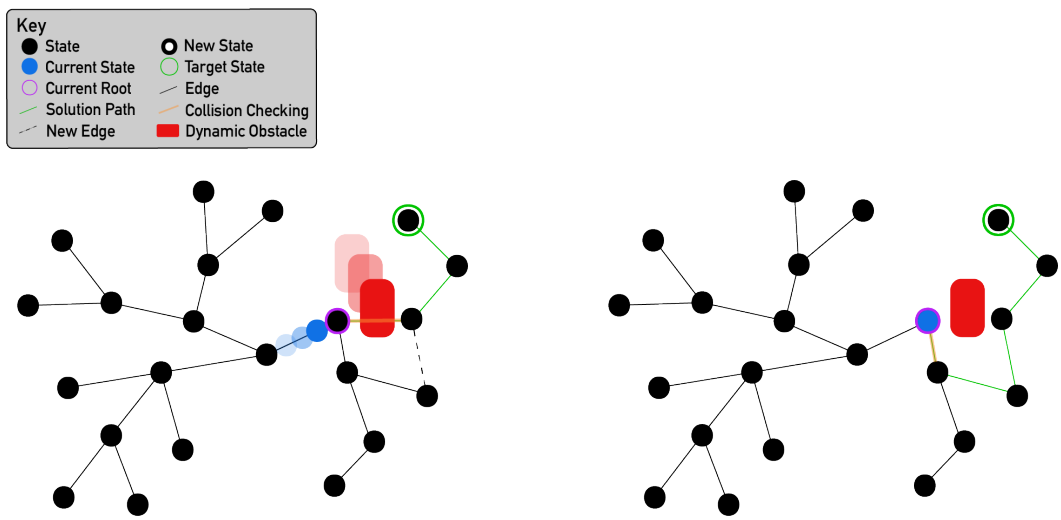
Figure 3.7: PRT-RRT* Planner States

path, $q_i \in Q_{target}$ in order to yield an unobstructed solution path to the target state among nodes already in the tree. If the reroute routine fails to find an unobstructed path to the target already in the tree, then the planner will expand the tree via RRT* until a solution path is found. The reroute routine is outlined in Algorithm 7 and an example of a successful reroute is depicted in Figure 3.8. This reroute routine is a novel contribution.

To ensure solution paths account for up-to-date information about the environment, the planner component receives information about the current state of obstacles $Q(t)_{obs}$ and the current target state $q(t)_{target}$ from the sensor component.

Algorithm 7: PRT-RRT* Reroute Routine

Input: $\mathcal{T}, Q(t)_{target}, Q(t)_{obs}$
for $q_i \in Q(t)_{target}$ **do**
  if $q_i \in Q(t)_{obs}$ **then**
    continue
  else
    $Q_{nn} = \text{GetNeighbors}(\mathcal{T}, q_i)$
    AddToBestParent$(\mathcal{T}, Q_{nn}, p_i, q_i)$ using Algorithm 3
  end
end
Output: $True$ if reroute succeeded, else $False$



(a) The solution path (green path) becomes obstructed by a dynamic obstacle (red block) and the planner attempts to reroute the next state in the solution path.

(b) The reroute is successful and an unobstructed solution path (green path) is found to the target state (green circle).

Figure 3.8: Example of Successful PRT-RRT* Reroute.

Hyperparameters. The hyperparameters incorporated in the PRT-RRT* planner are listed below.

- Max distance (r_{max}): Maximum euclidean distance between two connected nodes in the tree.
- Goal Bias (α): Small number in the interval $[0, 1]$ that influences how often the goal is sampled. See Equation 3.3

- Max neighbors (k_{max}): Maximum neighbors allowed within r_{max} of a node. If a new sample exceeds this number, then it will be rejected. See Algorithm 5.

- Nearest neighbor (r_{min}): Minimum Euclidean distance allowed between two nodes in the tree. If a new sample is closer than r_{min} to a sample already in the tree, then it will be rejected. See Algorithm 5

- Prime Tree seconds (t_{prime}): Amount of time in seconds the PRT-RRT* algorithm is initially allowed to plan before returning a solution.

3.5.2 Controller Component

The controller component executes solution paths $E_{target}(t)$ step-by-step after they have been received from the planner. Parallelizing this process allows the control signal frequency to be independent of the planner process frequency. Each step in the path must be explicitly declared collision-free by the collision checker before the controller will begin execution of the step.

As the controller component begins executing the next step in the current solution path, it outputs an update that it is executing the next step, and how long it will take to execute. If the controller component receives a notification from the sensor component that the target state has been updated or a notification from the collision checker that the next step in the path is obstructed, it will disregard its stored solution path and wait for the planner to output an updated solution path to the current target state before executing any more steps. The controller process is illustrated in Figure 3.9.

3.5.3 Collision Checker Component

The collision checker component repeatedly checks for collisions along the first edge in $E_{target}(t)$ between the current root node, $q_{root}(t)$ and the second node in $Q_{target}(t)$ $\forall t$. When the collision checker component first checks for collisions along a new edge, or when the collision status changes along an edge already checked for collisions, it

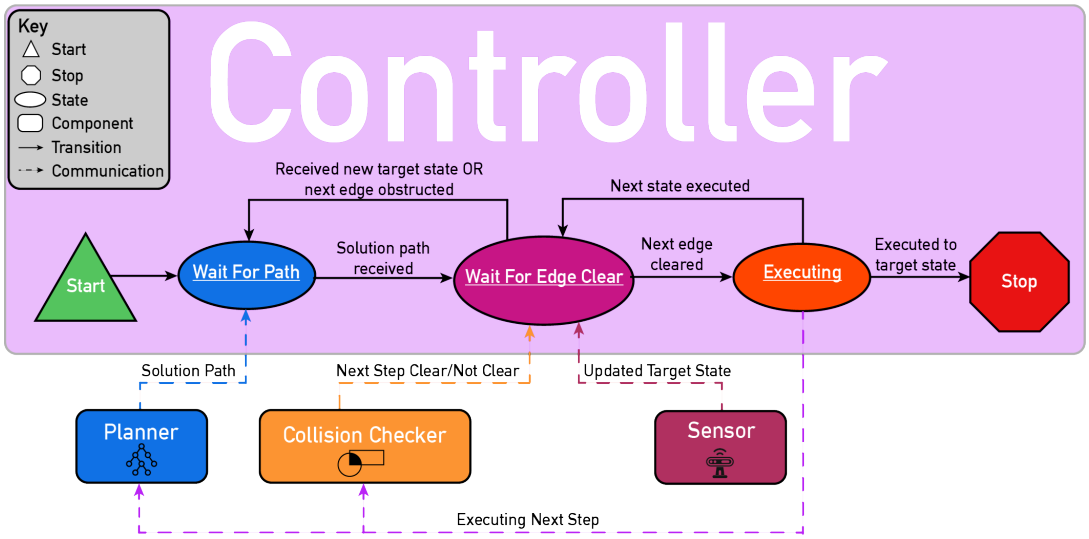


Figure 3.9: PRT-RRT* Controller States

will output whether the edge is collision-free or not. Recall that $q_{root}(t)$ is advanced to the next node in $Q_{target}(t)$ when the controller begins executing from the previous root. The collision checker process is illustrated in Figure 3.10.

3.5.4 Sensor Component

The sensor component observes when obstacles move or the target state changes and communicates this information to other components. The sensor component is purely an observer of the environment and does not take any input from other components.

A good example to illustrate the sensor component detection is a camera observing the space around a robot arm for a pick and place application. If the robot is tasked with picking up moving objects, then both the target state and the obstacle state related to the object change. The sensor component must detect this and notify the other components.

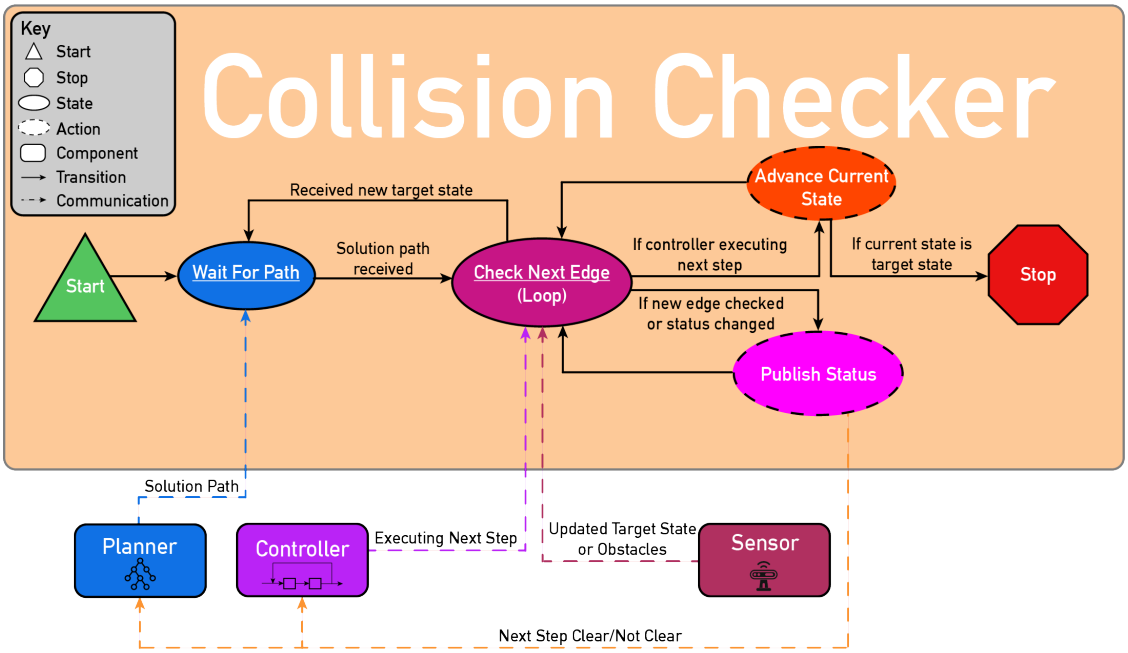


Figure 3.10: PRT-RRT* Collision Checker States

CHAPTER 4. SIMULATION METHODS AND RESULTS

The simulations were run on a Desktop PC with 20 Gb of RAM and an Intel Core i-5 CPU. Ubuntu 20.04, and ROS Noetic were installed on the Desktop. The path planning simulations were conducted for the 7-DoF Franka Emika Panda robotic arm.

4.1 Evaluation of RT-RRT* Real Time Performance for a 7D Robotic System

The Franka Emika Panda robotic arm hardware communicates data at a frequency of 1000 Hz, sending and receiving packets every 1 ms as outlined in [46]. If this control frequency requirement is violated then the robot will halt. High frequency control requirements are common for other 7DoF robotic arms as well like the Kinova Gen3 and the UR5 [47],[48].

Due to the serial execution of the tree maintenance and control sub-processes in the RT-RRT* algorithm, the combined processing time of each iteration must have an upper limit of 1 ms to satisfy the robot control frequency requirement. Thus if any one sub-process out of those performed in an RT-RRT* loop takes longer then 1 ms, then it follows that RT-RRT* is unable to perform under the control frequency constraint. While each sub-process can be manually limited in processing time, at a minimum the RT-RRT* loop must be able to complete at least one attempt to add a node to the tree and at least one rewiring to make progress. If the process is unable to attempt to add a node in each iteration, then it will be unable to explore the space and seek a solution path to a target state. If the process is unable to rewire at least one node in each iteration, then it will be unable to maintain the tree and keep the path planning data relevant to the agent as it moves. See Figure 3.4 for an example of an unmaintained vs. a maintained path planning tree.

The rewiring operation is the most complex sub-process within the RT-RRT* main loop, so to evaluate the feasibility of running the RT-RRT* loop in under 1 ms, data

was extracted on the RT-RRT* rewiring method. The rewiring method is outlined in Algorithm 4.

During simulation with the Panda robot arm, the sensor component published a new target state and the PRT-RRT* process planned and executed a solution path to the target state. Rewiring around the advancing root node occurred while the controller executed each step in the solution path. The average rewiring time per node was collected each time the rewiring sub-process was executed. This simulation was run in an obstacle free environment 10 times using the PRT-RRT* process. The amount of time it took on average to complete one rewiring operation as well as the minimum time it took to complete one rewiring operation per simulation run was extracted. The PRT-RRT* process was employed for these simulations because it uses the same rewiring operation as RT-RRT*, but will not fail if the operation takes too long due to the ability of the parallel control process to communicate at the required control frequency. The results are shown in Table 4.1. See Table 2 in Appendix A for the hyperparameter values used during the simulations.

Table 4.1: RT-RRT* Single Node Rewiring Time

Simulation Number	Avg. Rewire Time (msec/node)	Min. Rewire Time (msec/node)
1	2.534	1.074
2	2.302	1.683
3	2.631	1.842
4	2.392	1.206
5	2.708	1.701
6	3.114	2.347
7	2.353	1.781
8	2.269	1.752
9	2.155	1.537
10	3.031	1.903

Zero nodes were rewired in <1 msec, a requirement for the RT-RRT* algorithm inner-loop control to be feasible for the Franka Emika Panda robot arm.

4.2 Evaluation of Root Rewiring Impact on Executed Path Cost

The results above from Section 4.1 imply that in order to implement RT-RRT* on a 7D robot like the Franka Emika Panda, the root rewiring process would need to be disabled for the control process to maintain the required frequency. We designed the following methods to evaluate if it is worthwhile to maintain a path planning tree if it cannot be maintained using a root rewiring process.

Monitored RRT*. For the purpose of comparison, we designed a monitored RRT* (M-RRT*) planner process. The M-RRT* process uses the same sensor, collision detector, and controller components as the PRT-RRT* process. However, the planner component used by M-RRT* uses generic RRT* to solve for solutions between the current state of the robot and the target state. Whenever a new path needs to be planned due to a dynamic obstacle or change in target state, the M-RRT* planner starts planning a path from scratch using RRT* rather than rewiring and updating a stored tree as is done in the PRT-RRT* process. M-RRT* also uses the same sampling density rejection method as PRT-RRT* (Algorithm 5). With all other algorithmic details being the same, we assume any statistically significant variation between the performance of the M-RRT* and PRT-RRT* processes is due to storing vs. not storing the tree. The M-RRT* planner process is illustrated in Figure 4.1.

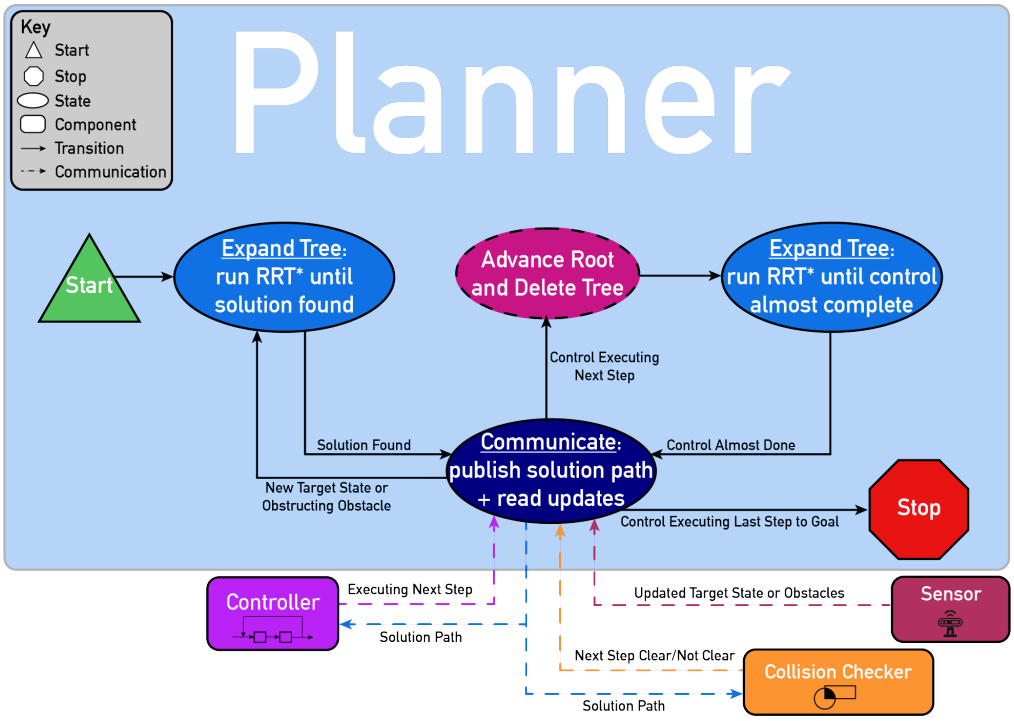


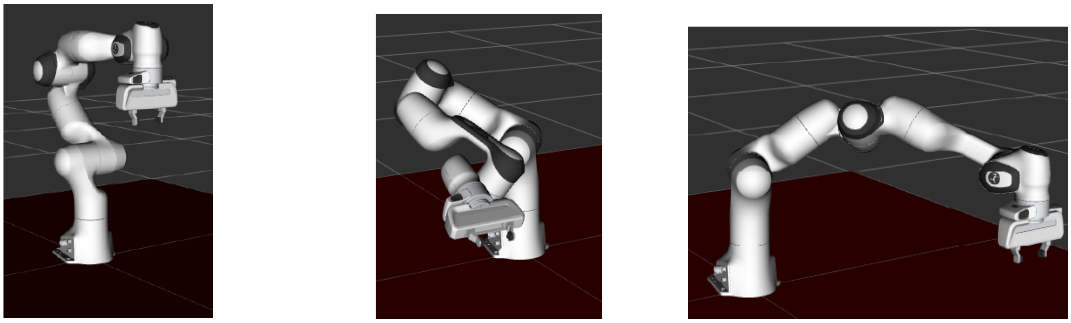
Figure 4.1: M-RRT* Planner Process

Primed PRT-RRT* No Rewire. The Primed PRT-RRT* No Rewire process is the same as the Primed PRT-RRT* process, except the root rewiring process is disabled. This is done to mimic the behavior of an RT-RRT* implementation for the 7D Panda configuration space, in which the process would be unable to rewire from the root and send control signals out at the required frequency.

Three planner processes were compared: the Primed PRT-RRT* planner process, the Primed PRT-RRT* No Rewire planner process, and the M-RRT* planner process. In the Primed PRT-RRT* processes, the planner was initially given five seconds to expand a planning tree without knowledge of the target or obstacle states. This allowed us to mimic maintaining a stored planning tree from before the target and obstacle states were published. After the initial five seconds had elapsed for the Primed PRT-RRT* processes and at the beginning of each simulation for M-RRT*, an initial target joint state was published by the sensor component. Two seconds after the initial target state was published, in which time the processes found and

began executing a solution path to the initial target joint state, a final target joint state was published.

The starting, initial target, and final target joint states are depicted in Figure 4.2. A total of 150 simulations were run, 50 using the Primed PRT-RRT* process, 50 using the Primed PRT-RRT* No Rewire process, and 50 using the M-RRT* process. The simulations were completed once the Panda was controlled to the final target joint state. The cost of the executed path was collected for each run and the results are shown in Table 4.2 and Figure 4.3.



(a) Starting Joint State (b) Initial Target Joint State (c) Final Target Joint State

Figure 4.2: Root Rewiring Evaluation States

Table 4.2: PRT-RRT* With vs. Without Root Rewiring Path Cost Results

Planning Process	Path Cost† Mean (μ)	Path Cost† Standard Dev (σ)
Primed PRT-RRT* No Rewire	7.159	1.071
Primed PRT-RRT* No Rewire	11.243	1.717
M-RRT*	8.327	3.725

N=50 samples were collected for each process (total 100 Samples).
Results showed the Primed PRT-RRT* and M-RRT* processes produced significantly lower cost paths than Primed PRT-RRT* No Rewire process. The independent t-test results are presented in Appendix A in Table 3.
† The cost of the path executed from the start state to the target state.

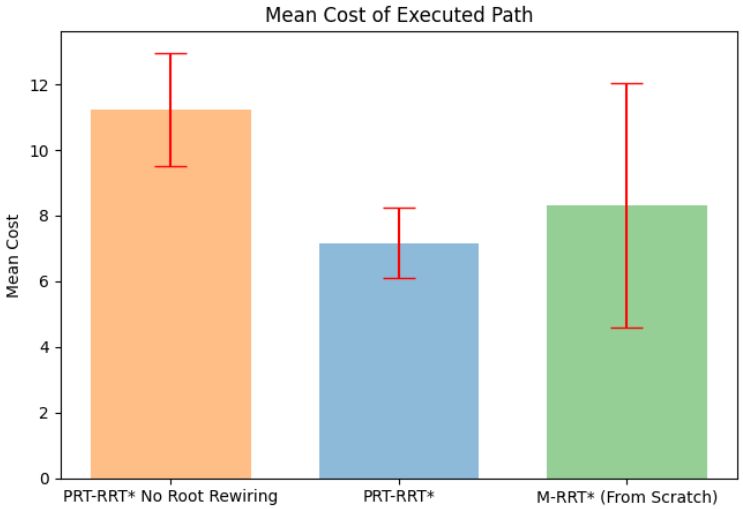


Figure 4.3: The Primed PRT-RRT* and M-RRT* processes executed significantly lower cost paths on average than the Primed PRT-RRT* No Rewire process.

4.3 Evaluation of PRT-RRT* Tree Maintenance vs. Planning From Scratch

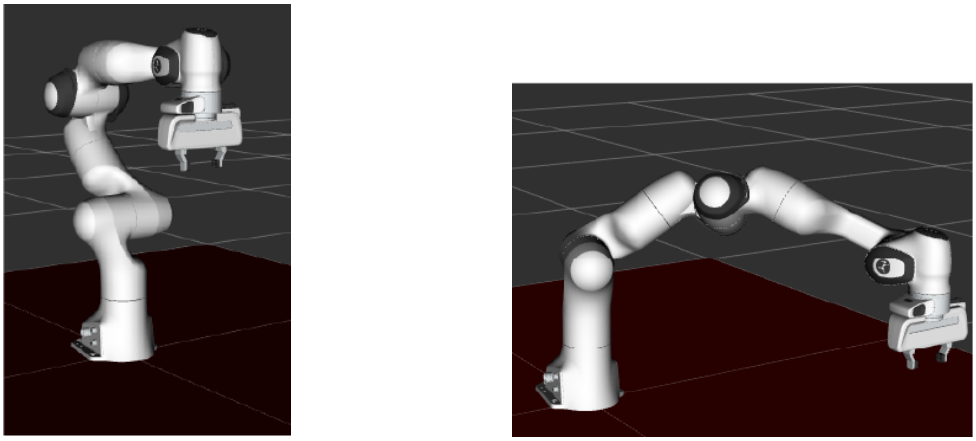
The results above from Section 4.2 show that without root rewiring, using a stored tree with Primed PRT-RRT* produces significantly higher-cost paths as opposed to re-planning from scratch with RRT* when changes occur in the environment. The following methods were designed to yield results that explore the value of maintaining a path planning tree with root rewiring compared to re-planning from scratch when changes occur in the environment.

4.3.1 Comparative Scenarios

Three scenarios with varying environments were designed to be tested with a primed PRT-RRT* process and the comparative M-RRT* process. In the primed PRT-RRT* process, the process was initially given five seconds to expand a planning tree without knowledge of the target or obstacle states before beginning each scenario. Thus, PRT-RRT* was able to build an uninformed RRT* tree exploring the environment before the target or obstacle states were identified. At the beginning of each scenario, the

sensor component published scenario-specific target and obstacle state information so each planner process would begin seeking the target state while avoiding the obstacles. Each scenario was concluded when the robot reached the target state. See Table 2 in Appendix A for the hyperparameters used during the simulations.

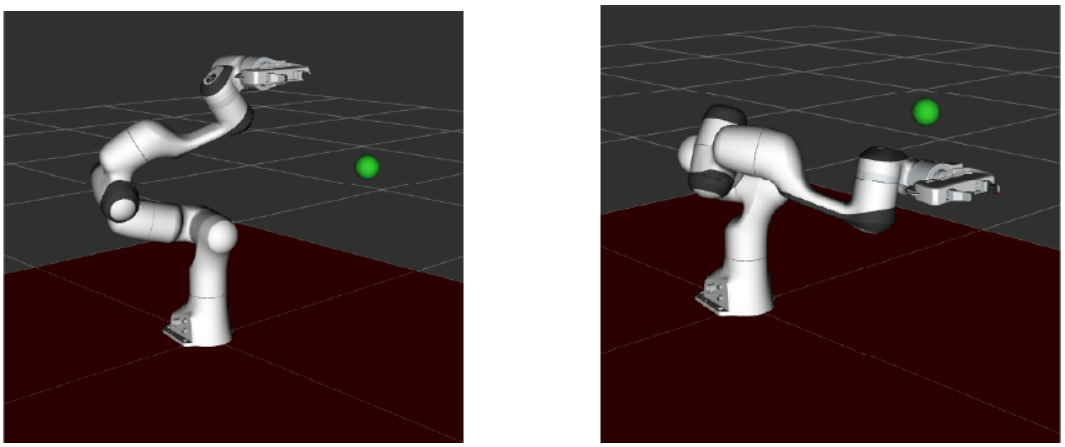
Scenario 1: No Obstacle. In the first scenario, a target state was set with no obstacles in the environment. The initial and target states are illustrated in Figure 4.4.



(a) Initial joint state (b) Target joint state

Figure 4.4: No Obstacle Scenario States

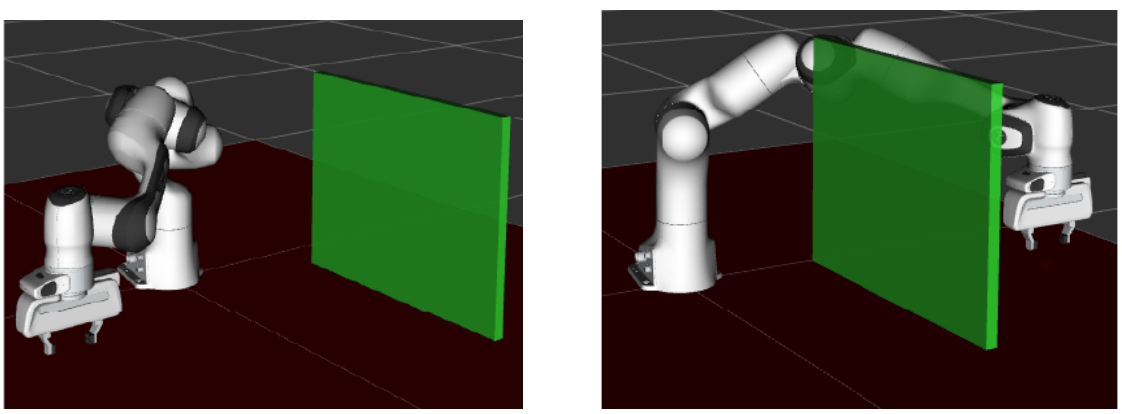
Scenario 2: Ball Obstacle. In the second scenario, a target state was set along with a ball obstacle that obstructed the shortest path between the robot start state and the target state. The initial and target states are illustrated with the ball obstacle in Figure 4.5.



(a) Initial joint state (b) Target joint state

Figure 4.5: Add Ball Scenario States

Scenario 3: Wall Obstacle. In the third scenario, a target state was set along with a wall obstacle that obstructed the shortest path between the robot start state and the target state and split the space in half. The initial and target states are illustrated with the wall obstacle in Figure 4.6.



(a) Initial joint state (b) Target joint state

Figure 4.6: Add Wall Scenario States

Each of the three scenarios were run in simulation with each of the two planning processes 50 times for a total of 300 runs. The cost of the path taken by the robot to reach the final target state in each run was collected. The average and standard deviation of all the samples for each scenario and process is displayed in Table 4.3.

The PRT-RRT* process required significantly less iterations to find a solution path to the target state in all three scenarios, and found significantly better paths in scenario 2 when compared with the M-RRT* process. The executed path cost results are plotted in Figure 4.7 and the initial solution iteration results are plotted in Figure 4.8.

Table 4.3: PRT-RRT* vs. M-RRT* Scenario Results

Scenario	Planning Process	Path Cost† $\mu \pm \sigma$	Iterations†† $\mu \pm \sigma$
1: No Obstacle	PRT-RRT*	6.324 ± 1.260	22.04 ± 18.19
	M-RRT*	6.741 ± 2.477	42.70 ± 29.10
2: Ball Obstacle	PRT-RRT*	5.498 ± 2.038	858.02 ± 3795.18
	M-RRT*	7.218 ± 3.381	15560.30 ± 17422.90
3: Wall Obstacle	PRT-RRT*	11.646 ± 6.635	24.98 ± 19.96
	M-RRT*	11.181 ± 3.359	554.80 ± 1110.40

N=50 samples were collected for each scenario and process (total 300 Samples). Results with significantly ($p < 0.05$) lower values are highlighted in green. The independent t-test results are presented in Table 4.

† The cost of the path executed from the initial state to the target state.

†† The number of expansion iterations taken to find an initial solution path to the target state.

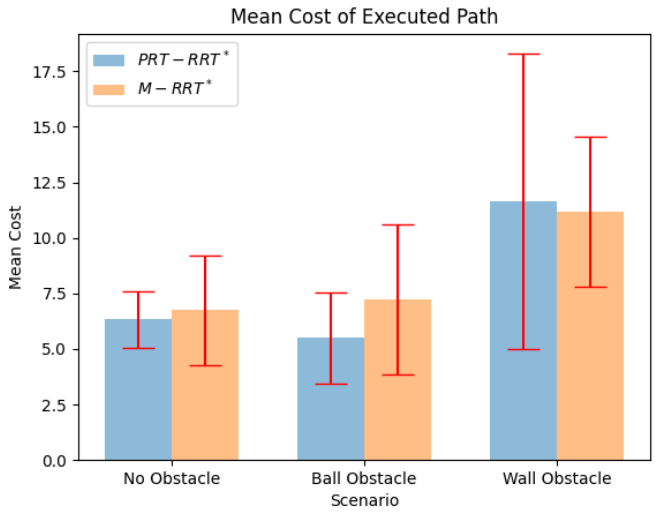


Figure 4.7: The Primed PRT-RRT* process and the M-RRT* process executed similar cost paths on average for the No Obstacle and Wall Obstacle scenario. And the Primed PRT-RRT* process executed significantly lower cost paths on average than the M-RRT* process for the Ball Obstacle scenario.

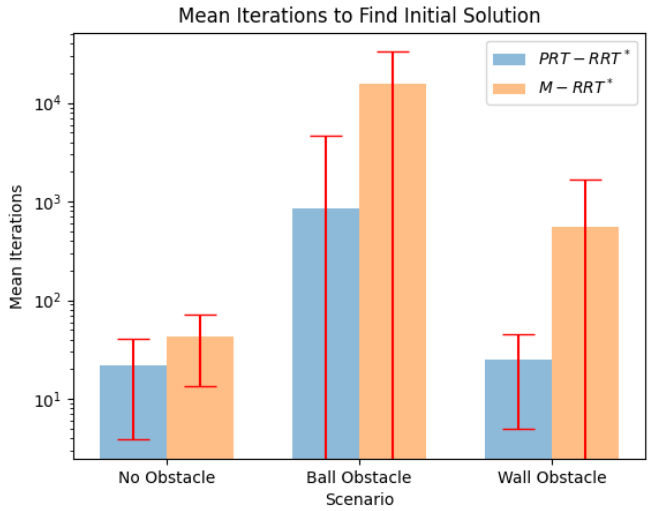


Figure 4.8: The Primed PRT-RRT* process found the initial solution to all scenarios in significantly less iterations on average than the M-RRT* process.

CHAPTER 5. DISCUSSION

5.1 Value of Parallezing RT-RRT*

The results displayed in Table 4.1 show that the RT-RRT* rewiring routine is unable to rewire a single node in the 7D path planning tree in under 1 msec. As discussed in Section 4.1, this is a hard requirement for RT-RRT* to be capable of executing at the necessary control frequency for the Panda robotic arm. Thus, if we wish to use RT-RRT* on a real robotic system like the Panda, then the root rewire routine must be disabled. The results displayed in Table 4.2 and Figure 4.3 show that the Primed PRT-RRT* No Rewire process yielded significantly higher cost executed paths than the M-RRT* process. This implies that we would be better off re-planning from scratch when changes occur in the environment as opposed to storing a path planning tree that is not rewired from the root.

The PRT-RRT* process we developed improves the efficiency of RT-RRT* such that root rewiring can be enabled in an 7D implementation on the Panda robotic arm while meeting the 1000hz control frequency constraint. In the PRT-RRT* process, the control component is able to maintain the control frequency on a separate thread, allowing the planner to take the required time to adequately maintain the tree on its own thread. The results displayed in Table 4.2 shows that with root rewiring enabled, storing and maintaining a tree produces similar executed path costs as re-planning from scratch.

A secondary benefit to the parallelization employed in the PRT-RRT* algorithm is the ability to easily plug in external controllers. The only constraint on a controller to work within the PRT-RRT* framework is that it must be able to control the agent within a neighborhood of a goal configuration in a known finite time. The RT-RRT* infrastructure lacks this ability because the control method is built into the planning loop and needs to be designed to work in serial with the tree maintenance methods.

5.2 Value of Preserving and Maintaining a Path Planning Tree

Iterations to find initial solution. As displayed in Table 4.3, in all three scenarios the primed PRT-RRT* planner was able to find an initial solution path to the target state in less iterations than M-RRT* on average. As shown in Table 4, these results were statistically significant. Based on this evidence, we conclude that building and maintaining a tree in an environment enables a planner to find solution paths more quickly and improves reactivity to changes in the environment. Note that some of the iteration data collected for scenarios 2 and 3 yielded high standard deviation due to outliers. During testing, the planner processes occasionally struggled to find solution paths around obstructing obstacles in some simulation runs, taking far more iterations to come up with an initial solution than the average. It is worth noting that while the M-RRT* results yielded high standard deviations in both scenarios 2 and 3, the PRT-RRT* results only yielded high standard deviation in scenario 2.

Executed path cost. Scenario 2 was the only scenario in which the executed path cost results was statistically significant when compared via an independent t-test (Table 4). The primed PRT-RRT* process executed significantly lower cost paths in scenario 2 as compared to the M-RRT* process. This result implies that for some situations, maintaining a pre-built tree may lead to lower-cost solution paths. However, for the other two scenarios, neither planning process executed significantly lower cost paths. Considering the nature of the dynamic path planning problem, there is no way to know in advance what tree will be most advantageous to build and store without any prior knowledge of obstacle or target states. Thus while storing a tree may sometimes lead to lower-cost path solutions, a pre-built tree may also contain many invalid or irrelevant paths once target and obstacle states change.

5.3 Limitations

5.3.1 Current Edge Obstructions

The collision checker employed in the PRT-RRT* (and RT-RRT*) process only checks for collisions between the current root and the next state in the current solution path. Since the agent is always being controlled to the current root, the current edge the robot is traveling along does not get checked for collisions. Thus, if a dynamic obstacle obstructs the motion of the robot along the current edge, then it will not be detected and the robot will run into it.

The current design of PRT-RRT* (and RT-RRT*) has no method to handle immediately obstructing obstacles while the robot is moving between solution path steps. Detection of such an obstruction is not a complicated operation, but the reaction requires some innovation. While a robot is traveling in between states in a solution path, there is no corresponding point in the path planning tree to represent the current state of the robot. Only once the robot arrives at the next state will the information maintained by the planner be relevant to the robot.

5.4 Future Work

5.4.1 Add Ability to React to Current Edge Obstructions

It is trivial to alter the collision checker component to check along the current edge for obstructions as noted in Section 5.3.1, so here we discuss potential reactions.

Stop in place. A potential solution is to notify the controller of a current edge obstruction and stop the robot in place. The planner could then add a node at the current state the robot is stopped at, and attempt to connect the node to the rest of the planning tree as the new root.

Incorporate alternative control methods. Considering the minimal constraints placed on the controller in the PRT-RRT* process, another potential solution would be to rely on an alternative controller to move away from nearby obstacles. A reinforcement learning, potential field, or control barrier function-based controller could

be employed to control the robot away from an approaching dynamic obstacle and towards a nearby node in the planning tree when a current edge obstruction is detected. Once the robot state aligns with a state in the path planning tree, the current root can be set and rewired to update the tree to the new current state of the robot.

5.4.2 Further Parallelization

In [18] and [19], the authors introduce parallelization techniques for the algorithms used within the planner and the collision checker. While the work presented in this thesis parallelizes the higher-level planner, controller, and collision checker components, for an even more efficient PRT-RRT* process, we propose implementing these lower-level parallelization techniques within the planner and collision checker components.

APPENDICES

Appendix A

Table 1: Panda PID Controller Gains

Joint	Proportional Gain	Derivative Gain	Integral Gain
q_1	600	30	0
q_2	600	30	0
q_3	600	30	0
q_4	600	30	0
q_5	250	10	0
q_6	150	10	0
q_7	50	5	0

Table 2: Simulation Hyperparameters

Hyperparameter	Value
Max distance (r_{max})	3.0
Goal bias (α)	0.05
Max neighbors (k_{max})	100
Nearest neighbor (r_{min})	0.1
Prime Tree seconds (t_{prime})	5.0

Table 3: Root Rewiring Evaluation Independent T-Test Results

Planner Process 1	Planner Process 2	t-value	p-value
PRT-RRT*	PRT-RRT* No Rewire	-6.652	1.664e-09
M-RRT*	PRT-RRT* No Rewire	-4.335	3.424e-05
PRT-RRT*	M-RRT*	-0.394	0.674

A negative t-value means lower executed path cost for the Planner Process 1 column.

p-values less than 0.05, which imply statistical significance, are highlighted in green

Table 4: Tree Maintenance vs. Planning From Scratch Independent T-Test Results

Comparison Variable	t-value	p-value
Scenario 1 Mean Cost	-1.0597	0.2919
Scenario 2 Mean Cost	-3.0816	0.0027
Scenario 3 Mean Cost	0.4426	0.6590
Scenario 1 Mean Iterations	-4.2585	4.7245e-05
Scenario 2 Mean Iterations	-5.8302	7.1032e-08
Scenario 3 Mean Iterations	-3.3733	0.0011

A negative t-value means lower cost/iterations for PRT-RRT* compared to M-RRT*

p-values less than 0.05, which imply statistical significance, are highlighted in green

Appendix B

RRT Rapidly exploring Random Trees: An iterative process that uses random sampling to quickly build a space-filling tree in search of feasible solution paths. 7–9, 11–13

RRT* Rapidly exploring Random Trees*: Optimal extension of RRT algorithm, using rewiring and a method to add new nodes to the best possible parent in the tree to ensure only optimal paths are included in the planning tree. 1, 8, 13–16, 21–23, 30, 33

RT-RRT* Real-Time RRT*: Extension of the RRT* algorithm that maintains and updates a path planning tree while controlling an agent along collision-free paths to a target state. If the process runs at a high enough frequency, RT-RRT* can be used in environments with moving obstacles and varying target states to quickly find new solution paths when changes occur in the environment. 1, 2, 8, 15–17, 22, 28–31, 38, 40

PRT-RRT* Parallel Real-Time RRT*: Parallelization of the RT-RRT* algorithm that enables high frequency control, collision checking, and environment sensing sub-processes to run in parallel with the more computationally expensive tree maintenance and expansion operations in the planner sub-process. This takes control frequency constraints off of the planning operations which is important when implementing tree maintenance operations in higher-dimensional spaces. 1, 2, 8, 20, 22, 24, 25, 29, 30, 33, 36, 38–41, 43

M-RRT* Monitored RRT*: Planning process that uses the same parallel sensor, collision checker, and controller sub-processes as PRT-RRT*, but only uses RRT* for the planning sub-process. M-RRT* throws the planning tree away and re-plans from scratch using RRT* in response to changes in the environment using rather than maintaining a tree as is done in PRT-RRT*. 30–33, 36–39, 43

Primed PRT-RRT* The PRT-RRT* process is initially given some time to expand a planning tree without knowledge of the target or obstacle states. This mimics holding on to and maintaining a planning tree for a long-running process. 31–33, 37

Primed PRT-RRT* No Rewire The Primed PRT-RRT* process where no root rewiring is allowed. This mimics the limitations of the RT-RRT* process for a 7D Implementation with control frequency constraints in which there is not enough time to maintain the planning tree effectively and send control signals at the required rate on a single thread. 31–33, 38

BIBLIOGRAPHY

[1] K. Naderi, J. Rajamäki, and P. Hämäläinen, "Rt-rrt*: A real-time path planning algorithm based on rrt*," in Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games, ser. MIG '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 113–118. [Online]. Available: https://doi.org/10.1145/2822013.2822036

[2] D. Yackzan. (2023) Panda robot arm demo. [Online]. Available: https://github.com/dwya222/robo_demo_ws

[3] J. Thumm and M. Althoff, "Provably safe deep reinforcement learning for robotic manipulation in human environments," in 2022 International Conference on Robotics and Automation (ICRA). IEEE, 2022, pp. 6344–6350. [Online]. Available: https://arxiv.org/abs/2205.06311

[4] J. Choi, G. Lee, and C. Lee, "Reinforcement learning-based dynamic obstacle avoidance and integration of path planning," Intelligent Service Robotics, vol. 14, no. 5, p. 663–677, 2021. [Online]. Available: https://doi.org/10.1007/s11370-021-00387-2

[5] O. Khatib, "The potential field approach and operational space formulation in robot control," Adaptive and Learning Systems: Theory and Applications, pp. 367–377, 1986. [Online]. Available: https://doi.org/10.1007/978-1-4757-1895-9_26

[6] Y. Koren and J. Borenstein, "Potential field methods and their inherent limitations for mobile robot navigation," in Proceedings. 1991 IEEE International Conference on Robotics and Automation, 1991, pp. 1398–1404 vol.2. [Online]. Available: https://doi.org/10.1109/ROBOT.1991.131810

[7] M. Davoodi, A. Iqbal, J. M. Cloud, W. J. Beksi, and N. R. Gans, "Safe robot trajectory control using probabilistic movement primitives and control barrier functions," Frontiers in Robotics and AI, vol. 9, Mar 2022. [Online]. Available: https://doi.org/10.3389%2Ffrobt.2022.772228

[8] Y. Chen, A. Singletary, and A. D. Ames, "Guaranteed obstacle avoidance for multi-robot operations with limited actuation: A control barrier function approach," IEEE Control Systems Letters, vol. 5, no. 1, pp. 127–132, 2020. [Online]. Available: https://doi.org/10.1109/LCSYS.2020.3000748

[9] G. Yang, B. Vang, Z. Serlin, C. Belta, and R. Tron, "Sampling-based motion planning via control barrier functions," in Proceedings of the 2019 3rd International Conference on Automation, Control and Robots. ACM, oct 2019, pp. 22–29. [Online]. Available: https://doi.org/10.1145%2F3365265.3365282

[10] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100–107, 1968. [Online]. Available: https://doi.org/10.1109/TSSC.1968.300136

[11] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," Journal of Artificial Intelligence Research, vol. 1, pp. 7–27, 1985. [Online]. Available: https://doi.org/10.1016/0004-3702(85)90084-0

[12] M. Likhachev and A. Stentz, "R* search," Association for the Advancement of Artificial Intelligence, Jul 2008. [Online]. Available: https://www.cs.cmu.edu/~maxim/files/rstar_aaai08.pdf

[13] R. E. Korf, "Real-time heuristic search," Artificial Intelligence, vol. 42, no. 2-3, pp. 189–211, Mar. 1990. [Online]. Available: https://doi.org/10.1016/0004-3702(90)90054-4

[14] J. Cannon, K. Rose, and W. Ruml, "Real-time motion planning with dynamic obstacles," Proceedings of the International Symposium on Combinatorial Search, vol. 3, no. 1, pp. 33–40, Aug 2021. [Online]. Available: https://doi.org/10.1609%2Fsocs.v3i1.18249

[15] A. Kushleyev and M. Likhachev, "Time-bounded lattice for efficient planning in dynamic environments," in 2009 IEEE International Conference on Robotics and Automation. IEEE, 2009, pp. 1662–1668. [Online]. Available: https://doi.org/10.1109/ROBOT.2009.5152860

[16] M. Phillips and M. Likhachev, "SIPP: Safe interval path planning for dynamic environments," in 2011 IEEE International Conference on Robotics and Automation. IEEE, May 2011. [Online]. Available: https://doi.org/10.1109%2Ficra.2011.5980306

[17] J. Snape, S. J. Guy, and J. van den Berg, "Independent navigation of multiple robots and virtual agents," in Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1, 2010, pp. 1645–1646. [Online]. Available: http://dx.doi.org/10.1145/1838206.1838522

[18] J. Pan, C. Lauterbach, and D. Manocha, "G-planner: Real-time motion planning and global navigation using gpus," in Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, ser. AAAI'10. AAAI Press, 2010, p. 1245–1251. [Online]. Available: https://doi.org/10.1609/aaai.v24i1.7732

[19] J. Pan and D. Manocha, "Gpu-based parallel collision detection for fast motion planning," The International Journal of Robotics Research, vol. 31, no. 2, pp. 187–200, 2012. [Online]. Available: https://doi.org/10.1177/0278364911429335

[20] S. Murray, W. Floyd-Jones, Y. Qi, D. J. Sorin, and G. D. Konidaris, "Robot motion planning on a chip." in Robotics: Science and Systems, vol. 6, 2016. [Online]. Available: https://doi.org/10.15607/rss.2016.xii.004

[21] S. Fujii and Q.-C. Pham, "Realtime trajectory smoothing with neural nets," in 2022 International Conference on Robotics and Automation (ICRA). IEEE, 2022, pp. 7248–7254. [Online]. Available: https://doi.org/10.1109%2Ficra46639.2022.9812418

[22] L. Janson, E. Schmerling, A. Clark, and M. Pavone, "Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions," The International journal of robotics research, vol. 34, no. 7, pp. 883–921, 2015. [Online]. Available: https://doi.org/10.48550/arXiv.1306.3532

[23] J. A. Sethian, "Fast marching methods," SIAM review, vol. 41, no. 2, pp. 199–235, 1999. [Online]. Available: https://www.jstor.org/stable/2653069

[24] J. A. Sethian and A. Vladimirsky, "Fast methods for the eikonal and related hamilton–jacobi equations on unstructured meshes," Proceedings of the National Academy of Sciences, vol. 97, no. 11, pp. 5699–5703, 2000. [Online]. Available: https://doi.org/10.1073/pnas.090060097

[25] J. Hou, Z. Liu, and H. Su, "Obstacle based fast marching tree for global motion planning," in IECON 2022 – 48th Annual Conference of the IEEE Industrial Electronics Society, 2022, pp. 1–6. [Online]. Available: https://doi.org/10.1109/IECON49645.2022.9968798

[26] M. Tukan, A. Maalouf, D. Feldman, and R. Poranne, "Obstacle aware sampling for path planning," in 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2022, pp. 13 676–13 683. [Online]. Available: https://doi.org/10.48550/arXiv.2203.04075

[27] Z. Chai and Z. Zhang, "Mobile robot path planning in 2d space: A survey," in 2022 International Symposium on Control Engineering and Robotics (ISCER), 2022, pp. 47–57. [Online]. Available: https://doi.org/10.1109/ISCER55570.2022.00015

[28] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Dept. Comput. Sci., Iowa State Univ., Ames, IA, USA, Technical Report TR 98-11, Oct. 1998. [Online]. Available: http://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf

[29] C. Urmson and R. Simmons, "Approaches for heuristically biasing rrt growth," in Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453), vol. 2, 2003, pp. 1178–1183 vol.2. [Online]. Available: https://doi.org/10.1109/IROS.2003.1248805

[30] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," The International Journal of Robotics Research, vol. 30, no. 7, pp. 846–894, 2011. [Online]. Available: https://arxiv.org/abs/1105.1186

[31] B. Boardman, T. Harden, and S. Martínez, "Improved Performance of Asymptotically Optimal Rapidly Exploring Random Trees," Journal of Dynamic Systems, Measurement, and Control, vol. 141, no. 1, 08 2018, 011002. [Online]. Available: https://doi.org/10.1115/1.4040970

[32] A. P. Matthew Jordan, "Optimal bidirectional rapidly-exploring random trees," CSAIL, MIT, Technical Report MIT-CSAIL-TR-2013-021, 2013. [Online]. Available: http://hdl.handle.net/1721.1/79884

[33] A. H. Qureshi, K. F. Iqbal, S. M. Qamar, F. Islam, Y. Ayaz, and N. Muhammad, "Potential guided directional-rrt* for accelerated motion planning in cluttered environments," in 2013 IEEE International Conference on Mechatronics and Automation, 2013, pp. 519–524. [Online]. Available: https://doi.org/10.1109/ICMA.2013.6617971

[34] A. H. Qureshi, S. Mumtaz, K. F. Iqbal, B. Ali, Y. Ayaz, F. Ahmed, M. S. Muhammad, O. Hasan, W. Y. Kim, and M. Ra, "Adaptive potential guided directional-rrt*," in 2013 IEEE International Conference on Robotics and Biomimetics (ROBIO), 2013, pp. 1887–1892. [Online]. Available: https://doi.org/10.1109/ROBIO.2013.6739744

[35] A. H. Qureshi and Y. Ayaz, "Intelligent bidirectional rapidly-exploring random trees for optimal motion planning in complex cluttered environments," Robotics and Autonomous Systems, vol. 68, pp. 1–11, 2015. [Online]. Available: https://doi.org/10.1016%2Fj.robot.2015.02.007

[36] P. Xin, X. Wang, X. Liu, Y. Wang, Z. Zhai, and X. Ma, "Improved bidirectional rrt* algorithm for robot path planning," Sensors, vol. 23, no. 2, 2023. [Online]. Available: https://www.mdpi.com/1424-8220/23/2/1041

[37] Z. Tahir, A. H. Qureshi, Y. Ayaz, and R. Nawaz, "Potentially guided bidirectionalized rrt* for fast optimal path planning in cluttered environments," Robotics and Autonomous Systems, vol. 108, pp. 13–27, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889017309387

[38] J. Kuffner and S. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065), vol. 2, 2000, pp. 995–1001 vol.2. [Online]. Available: https://doi.org/10.1109/ROBOT.2000.844730

[39] S. Klemm, J. Oberländer, A. Hermann, A. Roennau, T. Schamm, J. M. Zollner, and R. Dillmann, "Rrt*-connect: Faster, asymptotically optimal motion planning," in 2015 IEEE International Conference on Robotics and Biomimetics (ROBIO), 2015, pp. 1670–1677. [Online]. Available: https://doi.org/10.1109/ROBIO.2015.7419012

[40] J. Wang, W. Chi, C. Li, C. Wang, and M. Q.-H. Meng, "Neural rrt*: Learning-based optimal path planning," IEEE Transactions on Automation Science and Engineering, vol. 17, no. 4, pp. 1748–1758, 2020. [Online]. Available: https://doi.org/10.1109/TASE.2020.2976560

[41] J. Wang, J. Liu, W. Chen, W. Chi, and M. Q.-H. Meng, "Robot path planning via neural-network-driven prediction," IEEE Transactions on Artificial Intelligence, vol. 3, no. 3, pp. 451–460, 2022. [Online]. Available: https://doi.org/10.1109/TAI.2021.3119890

[42] J. Wang, X. Jia, T. Zhang, N. Ma, and M. Q.-H. Meng, "Deep neural network enhanced sampling-based path planning in 3d space," IEEE Transactions on Automation Science and Engineering, vol. 19, no. 4, pp. 3434–3443, 2022.

[43] M. Otte and E. Frazzoli, "Rrt x: Real-time motion planning/replanning for environments with unpredictable obstacles," in Algorithmic foundations of robotics XI: selected contributions of the eleventh international workshop on the algorithmic foundations of robotics. Springer, 2015, pp. 461–478. [Online]. Available: http://ottelab.com/html_stuff/pdf_files/Otte.Frazzoli.InSubmission.pdf

[44] M. Otte and Frazzoli, "Rrtx: Asymptotically optimal single-query sampling-based motion planning with quick replanning," The International Journal of Robotics Research, vol. 35, no. 7, pp. 797–822, 2016. [Online]. Available: https://doi.org/10.1177/0278364915594679

[45] O. Robotics. (2018) Joint trajectory controller. [Online]. Available: http://wiki.ros.org/joint_trajectory_controller

[46] F. Emika. (2017) Minimum system and network requirements - franka control interface (fci). [Online]. Available: https://frankaemika.github.io/docs/requirements.html

[47] Kinova. (2022) Kinova user guide. [Online]. Available: https://www.kinovarobotics.com/uploads/User-Guide-Gen3-R07.pdf

[48] U. Robots. Ur5 technical details. [Online]. Available: https://www.universal-robots.com/media/1802778/ur5e-32528_ur_technical_details_.pdf

VITA

David Yackzan

Education

- B.S. in Mechanical Engineering from the University of Dayton. December, 2020.

Professional positions held

- January 2021 to Present: Research Assistant
- May 2019 to Present: Robotics Software Engineer at Badger Technologies

Scholastic and professional honors

- National Science Foundation Graduate Research Fellowship Program: 2022 Honorable Mention
- Dean's List All Semesters