RRT Based Motion Planner for Robotic Arm Manipulator

Abhishek Anil Avhad
UID : 120257162
University of Maryland
abhi6@umd.edu

Gautam Sreenarayanan Nair
UID : 119543092
University of Maryland
gautamsn@umd.edu

Piyush Goenka
UID : 120189500
University of Maryland
pgoenka@umd.edu

***Abstract*—Motion planning in robotics involves finding feasible paths for robots to navigate through complex environments. Traditional methods, such as grid-based graph searches, face limitations in higher-dimensional spaces and may not guarantee optimality. This project explores the implementation of the Rapidly Exploring Random Trees (RRT) and RRT* algorithms for motion planning of a manipulator in three-dimensional space, focusing on pick-and-place tasks. RRT* stands out as an efficient solution for navigating high-dimensional spaces by systematically expanding a tree towards the target configuration using random sampling and local optimization. The algorithms are implemented and tested in a simulation environment using ROS2 and Gazebo, with results demonstrating the effectiveness of RRT* in generating feasible paths while avoiding obstacles. The study concludes by discussing future work to enhance the algorithm's robustness and its potential application in real-world scenarios.**

***Index Terms*—RRT star, RRT, Gazebo, ROS2, UR3e**

I. Introduction

Motion planning problems have been approached historically by discretizing the the state space by random sampling or through graph based grid searches. It is guaranteed that grid based graph search algorithms such as A* [3] provides optimal solution every time if it exists. However it's performance is highly depended on the map and discretization of the same. Also, they do not perform well with higher state dimensions [2]. In this scenario, the selected application involves planning for a manipulator. Due to the vast dimensional space involved, grid-based graph search algorithms were ruled out. Stochastic search algorithms are deemed more suitable for this application. Consequently, algorithms based on Rapidly Exploring Random Trees (RRTs) [5] were selected.

The Rapidly Exploring Random Tree (RRT*) algorithm stands out as a potent motion planning solution employed in robotics to navigate high-dimensional spaces with efficiency. It operates by systematically expanding a tree from the starting configuration toward the target configuration, employing random sampling and local optimization to identify viable paths. The goal of this project was to implement this algorithm for a manipulator operating in three-dimensional space, specifically for pick-and-place tasks. Manipulator robots serve diverse functions, ranging from mundane activities like fruit picking to intricate surgical procedures in operating theaters. By integrating this algorithm into the project, the objective is to streamline the motion planning process for these tasks, enhancing overall efficiency.

II. Literature Review

The prior work reviewed here primarily provide insights into solving the motion planning problem, which involves finding paths through graphs. [2] primarily focuses on sampling-based methods, particularly Rapidly-exploring Random Trees (RRTs), and introduces a modification called Informed RRT*. On the other hand, [3] discusses algorithms for finding minimum cost paths through graphs, thereby blending a good mix of mathematical and heuristic approaches.

Motion planning, which is a fundamental problem in robotics, often involves navigating through agile and complex spaces efficiently. Traditionally, this problem has been tackled using either mathematical or heuristic approaches, each of which has its own strengths and its own limitations.

The mathematical approach seen in [3], emphasizes the properties of abstract graphs and the development of algorithms ensuring that there is optimality in finding minimum cost paths. These algorithms, such as the dynamic programming-based methods, guarantee solutions for any given graph but may face computational challenges, especially with large-scale problems.

Contrasting, [2] delves into heuristic-based techniques, particularly sampling-based methods like RRTs, which avoid the need for explicit discretization of the state space. While these methods scale better with problem size and consider constraints such as kinodynamic constraints directly, regrettably they do not offer the same guarantees of optimality as their mathematical contemporaries.

A significant advancement presented in [2] is the introduction of Informed RRT*, a newer modification of RRT* that leverages the heuristic information to focus the search. This leads to faster convergence to optimal solutions. This approach strikes a balance between exploitation and exploration without additional parameter tuning, making it practical for real-world planning scenarios.

Furthermore, [2] addresses the limitations of traditional RRTs, which are shown to be sub-optimal. By introducing incremental rewiring of the graph, RRT* aims for optimality, but at the expense of increased computational complexity, especially in high dimensions.

In the broader context, both papers contribute to the ongoing evolution of motion planning algorithms. While [3] lays the groundwork for understanding optimal path-finding algorithms in graph-based scenarios, [2] advances the field by proposing a practical enhancement to sampling-based methods, bridging the gap between theory and practical application.

To sum it up, the literature that we reviewed highlights the complementary nature of mathematical and heuristic approaches in addressing the motion-planning problem. Moving forward, integrating insights from both paradigms could lead to more robust and efficient algorithms for navigating complex environments in various domains.

III. Algorithm

The Rapidly-exploring Random Tree (RRT) algorithm is a handy tool in robotics for motion planning. It helps robots figure out how to move from one spot to another. Essentially, RRT acts like a cartographer, mapping out potential routes by randomly selecting points in configuration space and extending the tree structure towards them. While RRT may not always find the globally optimal path, its strength lies in its ability to quickly navigate through high-dimensional and complex environments.

Pseudo code for RRT algorithm:

Step 1: Start
Step 2: Get Start and Goal points
Step 3: Generate a random point in the 3D workspace.
Step 4: Find the nearest available node from the random point.
Step 5: Create another point with a fixed step size from the nearest node in the direction of the random point.
Step 6: Check if new point and the path between it and nearest node fall in obstacle space.
If yes, then delete point. If not, then add point in list as a node with nearest node as parent.
Step 7: Go to Step 3
Step 8: Keep looping Step 3, 4, 5, 6 and 7 until we reach within a set threshold from goal point. In every 5 iterations of Step 3, generate a random point which is near to the goal point (bias).
Step 9: Once the goal is reached, perform backtracking to get the path

RRT* is an advanced iteration of the RRT algorithm. Its optimization lies in the continual refinement of the tree structure. This is achieved by dynamically reassessing and adjusting connections between newly added nodes and existing ones using cost function and rewiring. By minimizing the overall cost of traversing from the start to the goal configuration, RRT* seeks to find more optimal paths.

Pseudo code for RRT* algorithm:

Step 1: Start
Step 2: Get Start and Goal points
Step 3: Generate a random point in the 3D workspace.
Step 4: Find the nearest available node from the random point.
Step 5: Create another point with a fixed step size from the nearest node in the direction of the random point.
Step 6: Check if new point and the path between it and nearest node fall in obstacle space.
If yes, then delete point. If not, then add point in list as a node.
Step 7: Create a radius with fixed step size from new node and find the cost of all nodes that lie in the radius. Add the node with least cost as parent.
Step 8: Perform rewiring i.e. check if cost of nodes in radius reduces if connected to the new node. If yes, then make the new node the parent.
Step 9: Go to Step 3
Step 10: Keep looping Step 3, 4, 5, 6, 7,8 and 9 until we reach within a set threshold from goal point. In every 5 iterations of Step 3, generate a random point which is near to the goal point (bias).
Step 11: Once the goal is reached, perform backtracking to get the path

IV. Method

The initial step in implementation was to create a three dimensional space for the algorithm. This was implemented using python programming language and visualization was performed using the library Matplotlib. The next step was to create the workspace of robot (UR3e) for path planner. This was performed by using forward kinematics. Random sampling of nodes was done within this workspace. A cloud of random points is generated for visualization as shown in Fig. 6. While running the RRT and RRT* algorithm, the random nodes where taken from this cloud of points within the workspace of the manipulator used. The obstacles were added to the space for obstacle avoidance while planning the path. The path is generated using RRT or RRT* algorithm and the same is pruned to give a smoother path.The pruning is performed by skipping nodes in a particular threshold distance. The pruning threshold was chosen in such a way that it is lower than the clearance chosen so that node pruning doesn't cause obstacle collision. This was path generated is given as an input to the ROS2 nodes. The implementation is explained below.

The RRT* path planning algorithm was tested with a simulation setup of an UR3e robot attached with a vacuum gripper. The simulation framework chosen was ROS2 Humble with Gazebo simulator.

The UR3e robot is part of a series of cobots by Universal Robots. As seen in Fig 1 the UR3e is a 6-axis robot with a joint working range of ±360° on the first five axes and an infinite working range on the sixth axis. With a payload capacity of 3 kilograms, it has a maximum reach of 500mm. The robot comes with Force-Torque sensors on all axes. Thus, it is a very powerful and agile robot in a compact size.

Fig. 1. UR3e robot

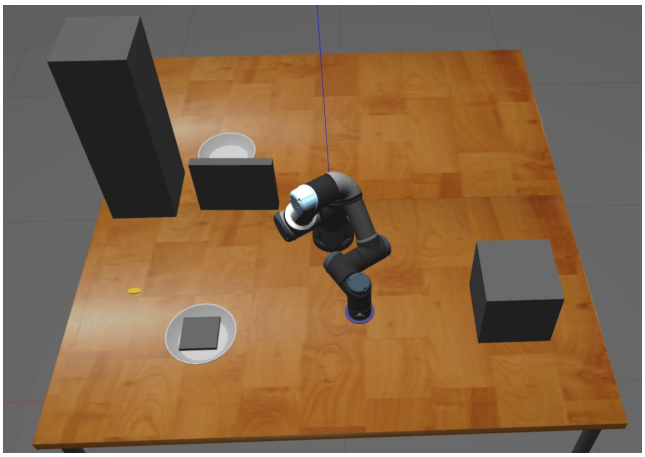

Fig. 2. Gazebo simulation setup

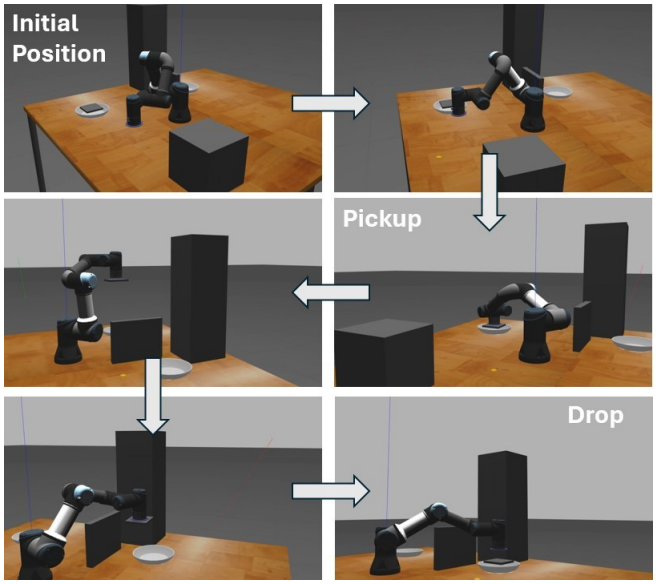

Fig. 3. Pick and Place operation

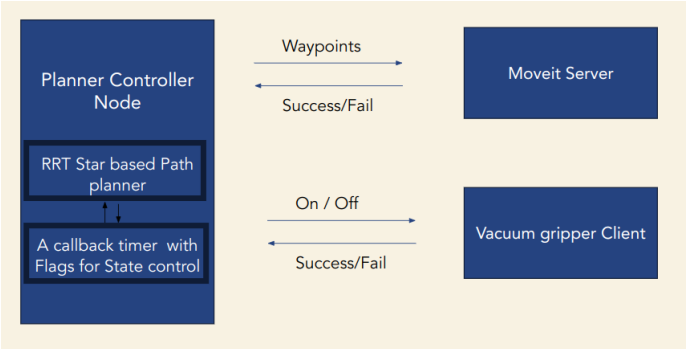

Fig. 4. Simulation Architecture

A custom Gazebo world was created where the UR3e robot was placed on a table as seen in 2. The table consists of three obstacles in the form of cuboids which are considered obstacles. Additionally, there are two white-colored plates. The goal of the robot is to pickup a tile from one plate and place it on the other plate while avoiding contact with obstacles. The vacuum gripper ensures that the tile is attached to the robot at all times when the robot is moving from one plate to another.

In Fig 3 it can be seen that the robot successfully goes to the pickup location, turns ON the vacuum gripper and traverses over an obstacle to finally reach the drop location. When at the drop location, the vacuum gripper turns OFF. At the end of this operation, the square tile has been successfully picked-up from one plate and placed on another.

To be able to generate the path and control the robot, we utilize ROS2 Humble architecture in conjunction to the Gazebo simulator. The architecture can be seen in Fig 4. We utilized the ROS2 client-server paradigm to achieve our tasks. We created two servers, one for path following and another to switch the vacuum gripper state between ON and OFF. The path/waypoint following server is a C++ ROS2 node which contains an object of MoveGroup class from moveit. We utilize the cartesian path following functionality of the MoveGroup class to move the robot in simulation. Our server accepts a PoseArray message from the client which consisits of an array of waypoints to be followed. We then send the waypoints information to the cartesian path planner and the cartesian path planner performs inverse kinematics and executes the motion. In case if the cartesian planner could not find the complete path, the server returns a fail message in the form of a boolean False. Else, it sends a success message in the form of a boolean True.

The main path planner controller node is the one which contains code for the logical flow of operations for successful pick and place of the tile. The logical data exchange between different nodes of the system can be visualised in Fig 4. The path planner node consists of an object of our developed RRT* algorithm. The algorithm takes in the start point location and goal point location as a list in the order of x,y and z. The units are in metres. The algorithm computes a path and provides a list of waypoints consisting of x,y and z positions. We send these waypoints to the waypoint server for execution. Similarly, during pickup and drop when the vacuum gripper needs to be turned ON and OFF respectively, the vacuum

gripper client sends the respective boolean value to the vacuum gripper server with true value indicating to turn ON the gripper and a false value to turn OFF the vacuum gripper.

V. Results

A. Time and Space complexities

Time and space complexities are important metrics used to study performances of algorithms. They are represented using big O notation [1] which describes the upper bound of a complexity. Time complexity refers to the amount of time an algorithm takes to complete as a function of the length of the input. It shows how the runtime of an algorithm grows with respect to the size of the input. Whereas, space complexity refers to the amount of memory space an algorithm uses as a function of the length of the input. Similar to time complexity. It measures how much additional memory is required by the algorithm for a size of input. The time and space complexities for RRT and RRT* algorithm are given below and the same is referenced from Dr.Sertac Karaman's study [4]. Essentially RRT and RRT* has same time and space complexity in big O notation as the upper bound for both are same. The same is given below.

- RRT
  Space Complexity: O(n)
  Time Complexity - Processing: O(n log n)
  Time Complexity - Query: O(n)

- RRT*
  Space Complexity: O(n)
  Time Complexity - Processing: O(n log n)
  Time Complexity - Query: O(n)

B. General Results

The simulation of UR3 in ROS2 and Gazebo yielded expected results. A comparison study was also performed between RRT and RRT Star algorithms within this specific environment. RRT and RRT Star being sampling based algorithms yield different paths in different runs. One of the negative aspects of these algorithms is that the path generated might not be the optimum path. However considering the higher dimensionality of the space in this application sampling based algorithms are better suited. It was a trade -off decision taken in this project. It can be seen that the specific problem here can generate to possible paths. One is over the obstacle as shown in Fig. 5. Another possible path is around the path as shown in Fig. 6. The algorithms could generate other possible paths too. However, in hundreds of iterations that the algorithm ran these were the two paths that the algorithm generated. Another point to be noted is that these paths are general directions. Even within these paths there could be minor variations due to it being sampling based.

Another minor study done parallelly for this project is the comparison between some metrics in RRT and RRT Star. These results are not generally relevant for this project however, the results generated are worthy for discussion. The first

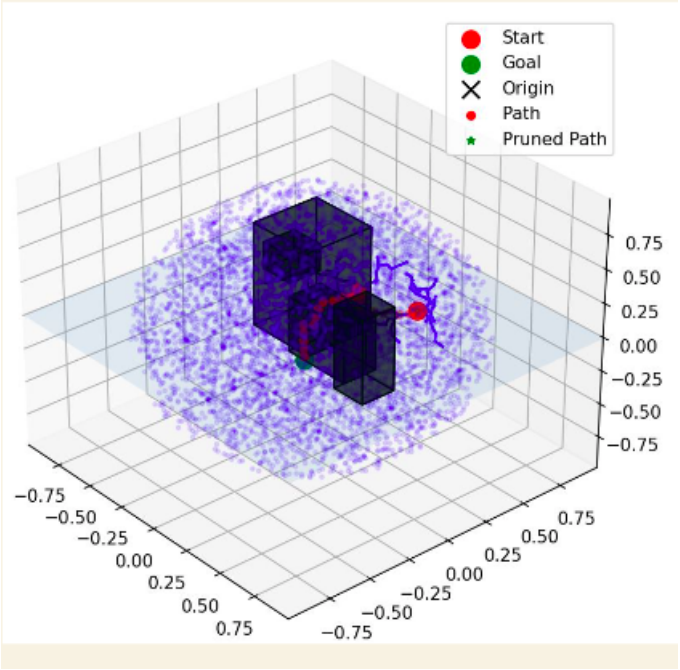

Fig. 5. Path over the obstacle

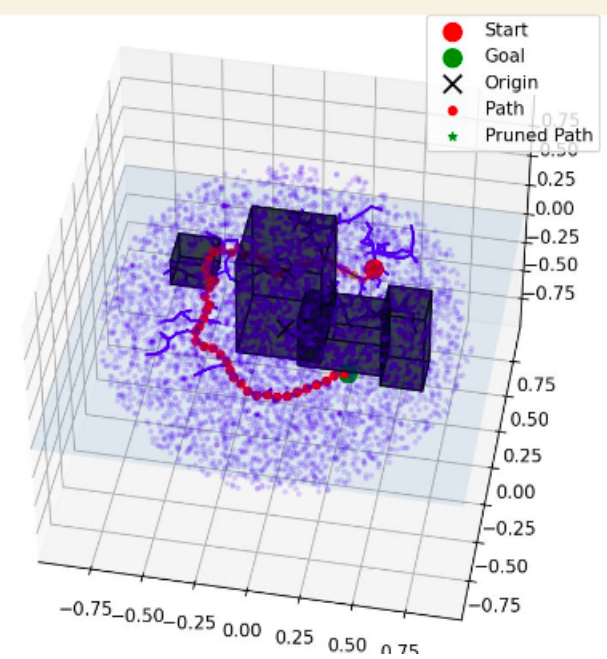

Fig. 6. Path around the obstacle

metric used is the CPU time (in seconds) taken for generating the path (Fig. 7). This path is the first path generated when the tree reaches the goal point threshold. CPU time is not the best method to measure the performance of an algorithm study but it can be a decent method of general comparison of algorithms. It can be seen that RRT star takes more time to generate a path than RRT. This result is expected because of additional

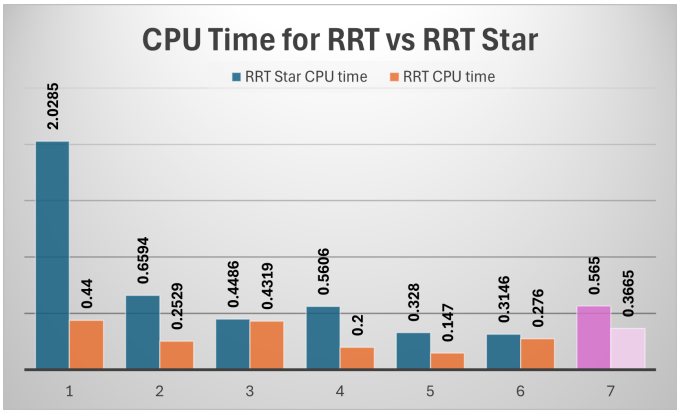

Fig. 7. CPU time (s)

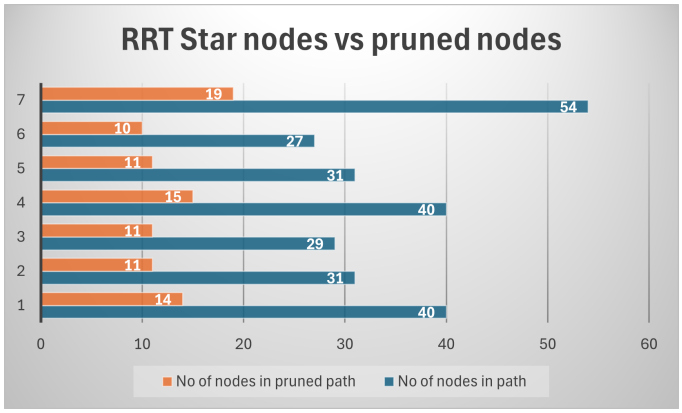

Fig. 9. RRT Star- Initial and pruned nodes

steps of rewiring that comes in RRT Star algorithm. Another interesting result is that the time taken for result generation is not directly depended on the path generated. i.e, a correlation cannot be drawn between the path generated and CPU time. This can be clearly seen in iteration number (7) of Fig. 7. This is time taken for generating path around the obstacle. However, a marked difference is not observed in time taken for path generation.

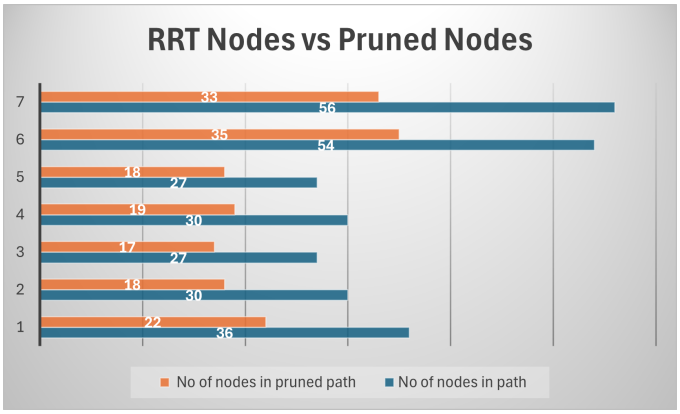

Fig. 8. RRT - Initial and pruned nodes

As mentioned earlier in this report, node pruning was performed on the path generated by the algorithm. This was generally done to smoothen and straighten the path in locations where nodes create unnecessary deviations from straight path. The pruning is performed by skipping nodes in a particular threshold distance. The nodes generated in path and pruned path of RRT is given in Fig. 8 and the same for RRT* is given in Fig. 9. Since pruning is done by skipping nodes within a particular distance, it can be noted that the node to pruned node ratio is higher for RRT* than RRT. This can be attributed to the rewiring aspect of RRT*. As RRT* performs rewiring , the nodes generated in path are closer-by generally than RRT. Another point to be noted is that the pruning threshold was chosen in such a way that it is lower than the clearance chosen so that node pruning doesn't cause obstacle collision. Another metric measured was the cost of RRT star for different iterations Fig. (8). This metric is not used for comparison but to measure the how optimized the path given by RRT* is. The shortest possible distance between pick-up and drop-points by avoiding obstacles is approximately 1.1 metres. The RRT* algorithm returned paths that are about 1.3 metres in length. This path is not optimum as expected.

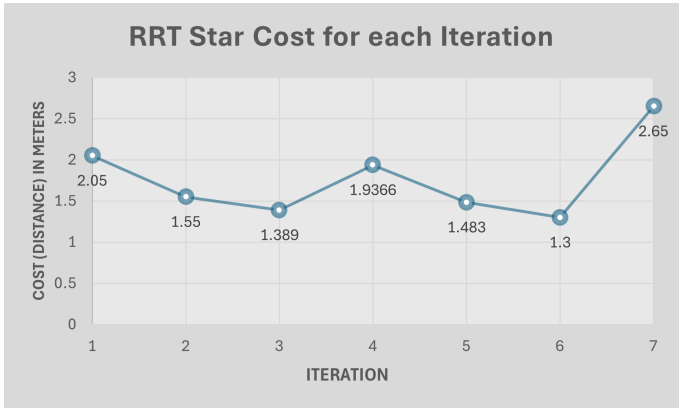

Fig. 10. RRT Star- Cost

VI. Conclusion

We used two widely used path planning algorithms RRT and RRT*. We found that RRT* outperforms RRT for our specific application. The design of our algorithm also tackled a variety of problems, one being kinematic constraints of the UR3e robot. We tackled it by adding a collision object in the space of the robot workspace where the robot would be in the "Up" configuration. This helped the path planner in avoiding to generate paths that would have gone through the axis of the robot. Another feature of our algorithm is obstacle avoidance, which can be seen by the fact that there is not path generated that goes through an obstacle. Another feature of our planner is being real-time, which helps to deploy it in

real world applications. Also, once the path is generated, we further optimise it to be smoother, which helps the robot to maintain a constant velocity.

In conclusion, we successfully generated efficient path for the UR3e robot's motion in 3D obstacle space and were able to perform pick and place operation while avoiding obstacles. Our future work includes implementing our algorithm on a physical UR3e robot setup and testing more path planning algorithms such as informed RRT*. We would also like to make our algorithm more robust in the future by incorporating inverse kinematics and self collision checking.

VII. Acknowledgement

We would like to acknowledge the guidance and support of our course instructor Dr. Reza Monfaredi and our course Teaching Assistants Mr. Saksham Verma and Mr. Shantanu Parab.

We would also like to acknowledge the contributions of developers and engineers in developing, maintaining and open sourcing the ROS2 framework and the Gazebo simulator, without which it would be very difficult to create robotic systems. Additionally, we would like to thank the developers from Universal Robots for maintaining and actively refining the universal robots ROS/ROS2 packages that contain packages with all UR robot descriptions, moveit configs, gazebo simulations, communication drivers and ROS2 drivers to interface with hardware. The availability of such packages helped us to reduce time on setting up the robot simulation and helped to focus on refining our algorithm.

The Universal Robots ROS2 packages can be found at https://github.com/UniversalRobots/

References

[1] I. Chivers and J. Sleightholme. *An Introduction to Algorithms and the Big O Notation*, pages 359–364. Springer International Publishing, Cham, 2015.

[2] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot. Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2997–3004, 2014.

[3] 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*, 4(2):100–107, 1968.

[4] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning, 2011.

[5] S. LAVALLE. Rapidly-exploring random trees : a new tool for path planning. *Research Report 9811*, 1998.