Trajectory Planning with Obstacle Avoidance
RRTs, A*, and R*

16-745 Optimal Control and RL - Spring 2020
Team: Into the Unknown

Parv Parkhiya
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
pparkhiy@andrew.cmu.edu

John Zucca
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213
jbzucca@andrew.cmu.edu

***Abstract*—Autonomous vehicles operating in the real world will not always know about all obstacles in the environment. Thus, the vehicle must have the ability to plan and re-plan trajectories around known obstacles and emerging obstacles in the environment. Currently, there are many trajectory search algorithms for autonomous driving vehicles. In this paper, we explore the most prominent trajectory search algorithms like RRT, A*, and R*. Our goal is to implement these algorithms to compare them with one another and determine their strengths and weaknesses. ADD SOME DISCUSSION ABOUT WHAT WORKED THE BEST We also implement a minor extension to efficiently update the RRT in case of new obstacle information without re-planning from scratch.**

I. Introduction

When selecting a trajectory planner for an autonomous vehicle, we must explore the trade-off between the creation of the optimal trajectory and the speed of trajectory generation. Both aspects of this trade-off are important. Usually getting an optimal trajectory takes extremely long time. If the trajectory generation speed is too slow, the vehicle may not be able to change its current trajectory in time to avoid a newly detected obstacle. Therefore we want a planner that gives optimum or close to optimum result in a reasonable time.

To ensure that both aspects of this trade-off are explored, three different search algorithms are explored in this paper. First, we chose a higher level trajectory planner, Rapidly Exploring Random Trees (RRTs). Then we chose a low-level trajectory planner, A*, which is guaranteed to achieve an optimum solution. We also look at weighted A* that gives early convergence at a cost of Optimality. Lastly, We explore the R* approach that's attempts to do better than weighted A* by mixing aspects of RRT and A*. This algorithm is not guaranteed to achieve the optimum solution on each created trajectory but is suppose to achieve it quickly. All algorithms are constrained in order to reflect the approximate dynamics of a bicycle model.

We began our investigations implementing all three of the above planners within an environment in which all obstacles are known. This was to evaluate the speed and accuracy of each of the planners to aid in planner selection for our adaptive environment planner. Once we determined which of the planners was best suited for our adaptive environment task, this planner was tested in our adaptive environment, which kept obstacles hidden from the planner and controller, until it was within a fixed distance from the vehicle.

II. Environment and Controller Details

The platform that we chose for evaluating this task was the Robot Operating System (ROS). Our environment consisted of a 20 meter by 20 meter plane of allowable space for our vehicle to traverse. We set up numerous goal positions, or waypoints, for our vehicle to obtain. These waypoints were defined using both (X, Y) coordinates and a vehicle direction, theta. These waypoints were fixed to only occur within a 20 meter by 20-meter plane at the center of the environment to prevent waypoints that would cause the vehicle to leave the 15 by 15 meters environment. waypoints were also generated such that they would not be present within any of the obstacles.

To properly test these algorithms, we wanted to present the planner with a combination of easy and difficult obstacle patterns. An example of the obstacle pattern that was used for testing can be seen in figure 1.

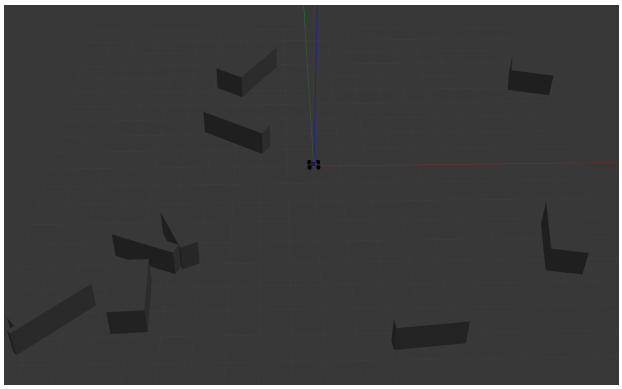

Fig. 1. Obstacle Environment

For the purpose of simulating a vehicle driving in this environment, we imposed the following assumed constraints on our vehicle that is modeled using bicycle dynamics:

- Wheelbase = $0.335m$
- Minimum turning radius = $0.67m$
- Maximum steering angle = $30degrees$
- Maximum velocity = $1m/s$
- Maximum acceleration = $4m/s^2$
- Maximum deceleration = $-4m/s^2$

A simple controller was created to follow trajectories in the environment in order to prove that the generated trajectories were feasible and avoided all obstacles. This controller supported both driving in forward and reverse directions at low speed. This mode of driving would most closely resemble driving in a parking lot, where low vehicle speeds are required. The controller implements a trajectory look-ahead to determine both the steering angle and whether the vehicle should be operating in reverse or forward gear. It's important to note that the focus of the project is the planning aspect and only a basic controller is implemented for completeness.

III. Implementation of Existing Algorithms

A. RRT Algorithm

1) General Algorithm Details: Rapidly exploring random tree (RRT) [1] as the name suggests rapidly explores the given space to find a feasible path. While extension like RRT* attempts to get close to optimal, RRT, in general, are not great in getting the optimal result. The expansion of the existing tree is shown in the fig 2. It involves the following steps

i) Sample a random point in the space
ii) From the existing tree structure find the closest point
iii) Consider the new node from the closest point in the direction of the sampled point
iv) Add the node if edge's intersection with the obstacle is null

Back pointer for every new node is also stored. The above steps are repeated in a loop till a node close enough to the goal is added in the tree and the entire path is constructed by recursively going back through the back pointer.

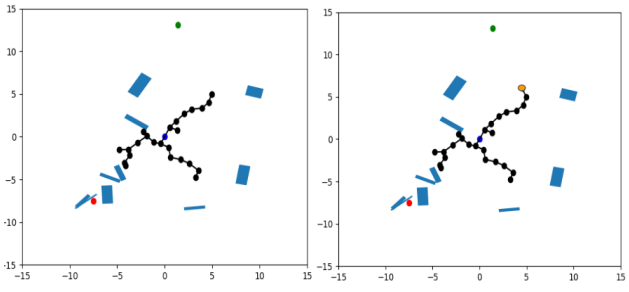

Fig. 2. Expanding a tree in RRT

RRT with nonholonomic constraints:

The path generated by naive RRT isn't necessarily a feasible path that could be followed by a vehicle with nonholonomic constraints. To account for these motion constraints, a slight modification would be required. Instead of extending the closest node trivially in the direction of the sampled node, multiple controls are randomly sampled and trajectories are rolled out from the closest node. The specific trajectory that brings the vehicle closest to the sampled point is chosen and corresponding control is also stored. The trajectory-rollout uses the vehicle motion model and hence generated path between nodes is always feasible.

2) Implementation Details: We implemented both the naive version of RRT as well as the RRT with the nonholonomic constraints from scratch. A minor trick to speed up the convergence of the RRT is to use bias sampling. While we don't want to sample all the points near the goal region to avoid getting stuck in local minima. From time to time, we do want the sampled point to be near the goal so that tree is more incentivized to grow towards the goal. Our implementation samples point in the goal region with some probability and remaining times uniformly in the arena. As shown in figure 3, after adding the goal bias the convergence happens quickly and the tree has far fewer nodes compare to without bias version. Having too high goal bias also leads to long convergence time because of getting stuck in local minima. Having 20% goal bias seemed to work well on the preliminary tests.

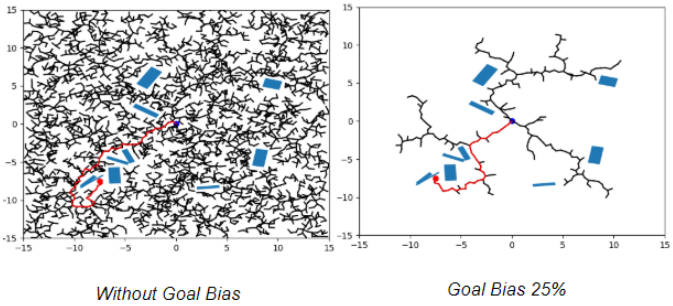

Fig. 3. RRT with and without goal bias sampling

We have used step length to be $1m$. We use an occupancy grid method to make sure all the nodes and edges are collision-free. For the RRT with nonholonomic constraints, we used 10 trajectory rollouts using randomly sampled allowable steering angles and 1-meter arc length. We have also added the ability to reverse the vehicle which is especially useful for tasks like parking in a confined space.

The costliest operation in the RRT algorithm is finding the closest node from the sampled node. A naive implementation would require comparing the sampled node with every existing node in the tree. But as the tree size grows, the time to find the closest node will also grow. We used kd-tree to efficiently find the closest node in the tree. Kd-tree works like a binary search but in k dimension. kd-tree brings down complexity from $O(n)$ to $O(log(n))$ where n is the existing number of nodes in the tree.

3) Results: Figure 4 shows the trajectory found by naive RRT as well as RRT with nonholonomic constraints. There is also a great deal of randomness associated with RRT and

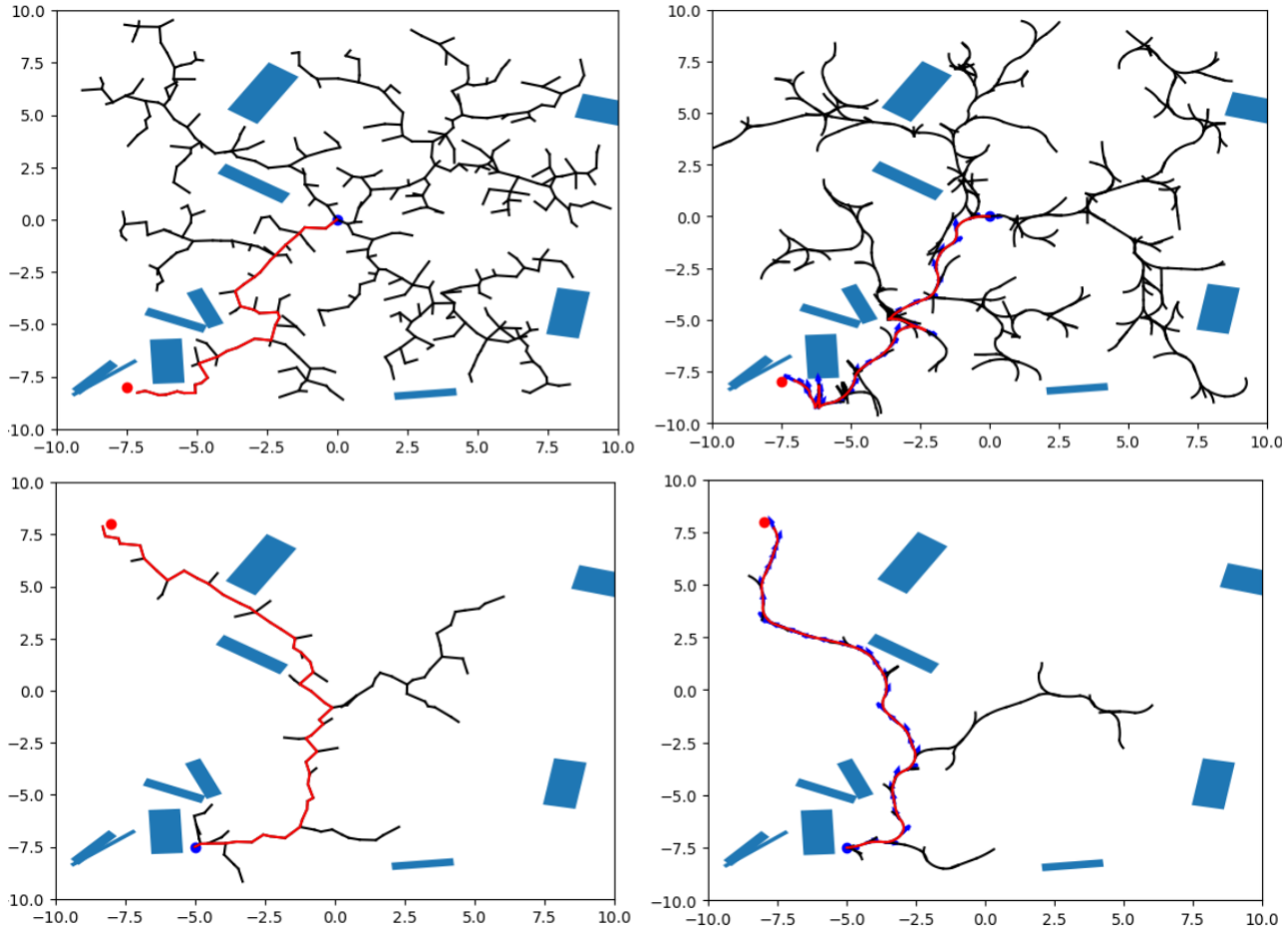

Fig. 4. Naive RRT (left) and RRT with nonholonomic constraints (right) from start(blue dot) to goal(red dot)

rerunning the algorithm produces different results every time. The figure has the same start and goal points indicated by blue and red points respectively. It's also important to note that the reverse functionality is enabled, so the path may seem not smooth at places but it's just switching between forward and reverse motion.

B. A* Algorithm

1) General Algorithm Details: The A* algorithm is implemented by representing the vehicle location and orientation in discrete states, or nodes. The first node is defined as the vehicle's current position at the time of planning. The last node in the trajectory will be the goal position. We maintain two lists of nodes for this algorithm, an open list, and a closed list. The open list represents the nodes that have been generated from a previously explored node but have not been fully explored themselves. The first node is initially included on the open list. The closed list represents the nodes that have been fully explored. When initialized, the closed list contains zero nodes.

Each node has two associated costs, the known cost, g, and the heuristic cost, h. The known cost is calculated by summing the costs to travel from node to node along the current path and it represents the true cost of traveling to the node from the origin node. The heuristic cost is an estimate of the remaining cost to achieve the goal position. A graphical representation of the two types of cost can be seen in figure 6. The formula for the heuristic cost is a design decision and the quality/efficiency of the algorithm is dependant on a quality heuristic function.

The algorithm is performed by choosing, from the open list, the node(s) which has the lowest total cost $f(s)$ to explore which is given by equation 1.

$$f(s) = g(s) + h(s) \tag{1}$$

Then, from this node, generate a list of valid successor nodes. These are nodes that the vehicle could traverse to next, so they should be constrained to only feasible next nodes. For example, in this environment, the successor nodes shouldn't violate any steering constraints or run into any obstacles. For each of the successor nodes, we then evaluate the true cost from the origin node by adding the g cost from the parent node and the cost to traverse from the parent node to the successor node. And we calculate the heuristic cost from our defined heuristic function. Once the costs have been calculated we add each node to the open list. If one of these nodes already exists on the open list, we replace the older

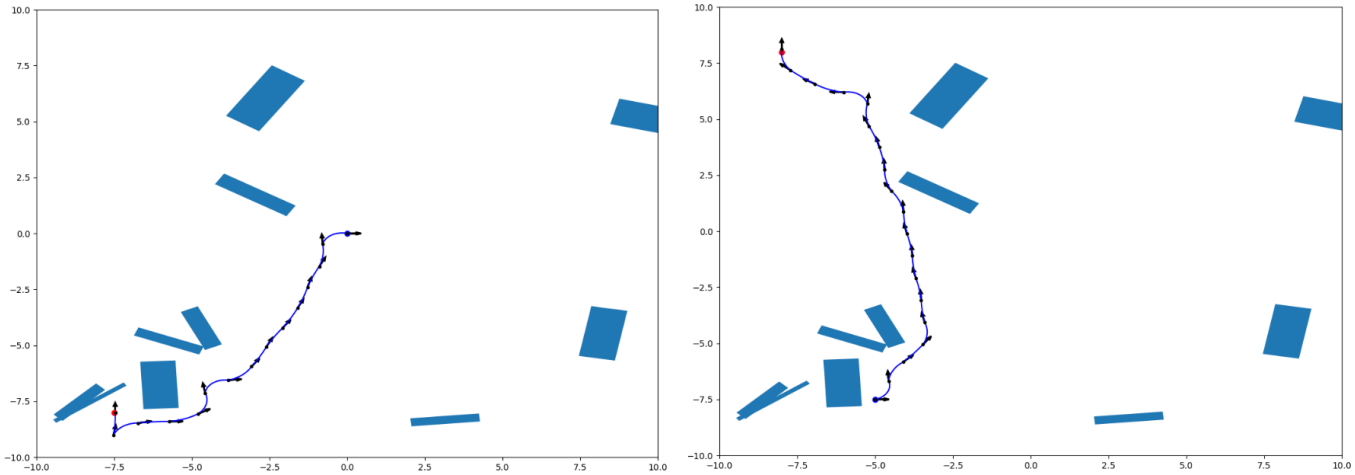

Fig. 5. Trajectories generated using weighted A* from start(blue) to goal(red)

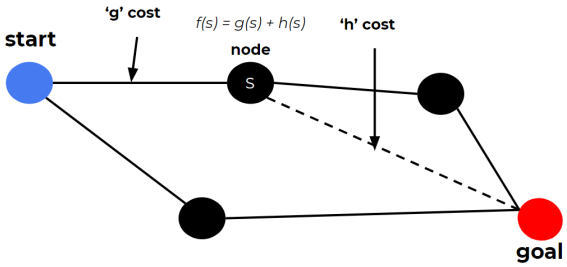

Fig. 6. Conceptual A* cost representation

node only if the calculated true cost, g, for the new node is lower than the g cost for the old node. This would mean that we have found a more efficient path from the start path to that node. Lastly, the parent node that has just been fully explored is added to the closed list. This loop is repeated until the goal has been reached.

2) Implementation Details: To set up the A* algorithm for this environment, we first had to determine how to discretize the environment. We explored two options. The first method that we utilized was creating a fixed (X, Y) grid, with .1 meters between each node in both the X and the Y directions. When creating the potential successor nodes, we chose to limit these nodes to those within .5 meters of the parent node, and remove all nodes that would violate the steering angle constraints of the vehicle or nodes that would violate an obstacle boundary. This method converged to a solution, but in doing so, it created a large number of nodes, took a long time to converge, and generated a jerky trajectory that would not be comfortable for a human passenger.

The second method for discretizing the environment is dynamic. We rollout trajectories with some arc length from the parent node to the successor node by sampling seven different steering angles distributed evenly within the steering angle constraints. These nodes obtained through rollout could be any continuous values and we store these values in Kd-tree. Before any new node is added, it's compared with the closest node in Kd-tree making sure and it's at least some fixed euclidean distance away. If it's too close to the existing node, it is merged with the closest node. This allowed us to reduce the number of states that we have to store apriori, while still allowing continuous state values and creating a smoother trajectory for the vehicle to follow.

The next design decision that we made in our implementation was our definition of the real cost function and the heuristic cost function. We defined the real cost function as the sum of the euclidean distance between all nodes in the trajectory. We determined that this would be a good cost function since it was most indicative of the length of time it would take our vehicle to follow the chosen path, due to the vehicle's constant speed. The heuristic function that we chose was also similar to Euclidean distance, however, we included the difference in vehicle orientation to force the vehicle to obtain the proper orientation to satisfy the requirements of the goal position. The heuristic function used is defined as follows: $h = \sqrt{(X_{goal} - X)^2 + (Y_{goal} - Y)^2 + (\theta_{goal} - \theta)^2}$.

Weighted A*

A* algorithm guarantees an optimal solution but that usually takes an extremely long time to converge. A simple solution to get early convergence at the cost of optimally is to do weighted A*. Weighted A* replaces total cost $f(s)$ from equation (1) to equation (2) where $w > 1$ is the weight parameter. Weighted A* guarantees solution with $w * OptimalCost$ To get early convergence we are using weighted A* with 2.0 weight value.

$$f(s) = g(s) + w * h(s) \tag{2}$$

3) Results: In order to easily recall the trajectory once the goal has been reached by the A* algorithm, we added the

previous node as part of the description of each node. This allowed us to reconstruct the trajectory by working backward from the goal position, adding each parent node to the front of the trajectory until the start node became the first node in the trajectory. The examples of the generated trajectories using our implementation of weighted A* can be seen in figure 5.

C. R* Algorithm

1) General Algorithm Details: Randomized A* search also known as R* [2] is an extension of the weighted A* algorithm that attempts to get early convergence while still maintaining sub-optimal bound of weight A* i.e. cost of the found solution will be less than or equal to w times the optimal cost.

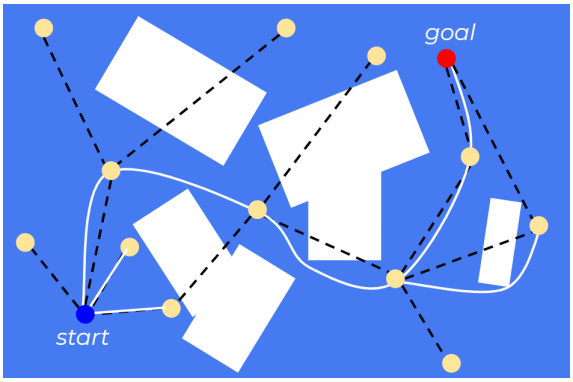

Fig. 7. conceptual R* graph

R* searches for the solution at 2 spatial scale level. The low level corresponds to an actual trajectory that vehicle can follow whereas high-level graphs correspond to meta graph with a bigger step size where there might or might not exist a feasible collision-free trajectory. The low-level graph is shown as a white curve in figure 7 whereas the high-level graph is shown with yellow nodes and dashed black edges. White rectangles are obstacles in the scene. The high-level graph acts as a randomized goal location that pushes the exploration towards the actual goal while still exploring other possibilities. Note that some of the edges in the high-level graph intersect the obstacles and not all low-level trajectory between high-level nodes is computed.

The high-level graph is also expanded with similar cost as equation (1) but here $g(s)$ also has heuristic component as an actual path to reach that node may not be computed yet. Nodes with the minimum total cost in the high-level graph are chosen for computing low-level graph in a bound manner i.e. after a certain time if the feasible path using weighted A* is not found, the algorithm marks the node as "avoid" and moves on the next node in the heap. Heap gives priority to the nodes that are not marked "avoid". Figure 8 details the pseudo-code for each iteration of the R* search.

2) Implementation Details: Like the RRT and A*, we also implemented the R* search algorithm from scratch. It uses a similar dynamic discretization method for both high-level and low-level graph as explained in the implementation of A*. A modified heap data structure is used to efficiently find the

1 select unexpanded state $s \in \Gamma$ (priority is given to states not labeled AVOID)
2 if path that corresponds to the edge $bp(s) \rightarrow s$ has not been computed yet
3 try to compute this path
4 if failed then label state s as AVOID
5 else
6 update $g(s)$ based on the cost of the found path and $g(bp(s))$
7 if $g(s) > w\, h(s_{\text{start}}, s)$ label s as AVOID
8 else //expand state s (grow Γ)
9 let $SUCCS(s)$ be K randomly chosen states at distance Δ from s
10 if goal state within Δ, then add it to $SUCCS(s)$
11 for each state $s' \in SUCCS(s)$, add s' and edge $s \rightarrow s'$ to Γ, set $bp(s') = s$

Fig. 8. Single iteration of R* search [2]

node with the smallest cost and kd-tree is used to dynamically merge with a close-by node.

Heap has been modified to consider the "avoid" flag along with the total cost to give proper priority. The algorithm also changes the $g(s)$ and "avoid" flag time to time which requires the update in the heap data structure which is a non-trivial task. We have implemented smart workarounds to add new cost information and discard outdated values in the heap.

High-level nodes are expanded by sampling points at Δ distance away with random orientation. Δ distance is chosen as $6m$. If the goal is less than Δ distance away from the current node, the goal node is also added in the high-level graph. For the low-level graph, weighted A* implemented in the previous section is used with one modification. If the number of nodes in the closed list for getting trajectory between two high-level nodes goes beyond a threshold than weighted A* search execution is stopped and returned back to R* handle. That threshold is chosen as a maximum of 1000 nodes in the closed list.

After convergence, the final trajectory from start to the goal is obtained by concatenating weighted A* trajectories between consecutive high-level nodes in the path.

3) Results: Like RRT and unlike A*, there exist a large amount of randomness is the algorithm. Specifically the sampling of the successor nodes in the high-level graph. Therefore re-running the algorithm with the same parameters produces a widely different result based on where high-level nodes are sampled.

Figure 9 shows the trajectory obtained using R* search along with the high-level graph. It's important to note that since trajectory obtained between nodes using weighted A* convergence only with some tolerance, the stitch together final trajectory is not strictly continuous. The discontinuity is closely related to the convergence criteria threshold. The figure linearly interpolates between these minor discontinuities.

IV. Comparison Between Different Methods

To compare the results of each method, we chose to record the average path length and completion time on the same course. The course consisted of five waypoints and 10 immobile objects. We have used the same gazebo environment as figure 1. The symbolic depiction of that environment with waypoints can be seen in figure 11. We also subjectively viewed the generated trajectories and rated each method on

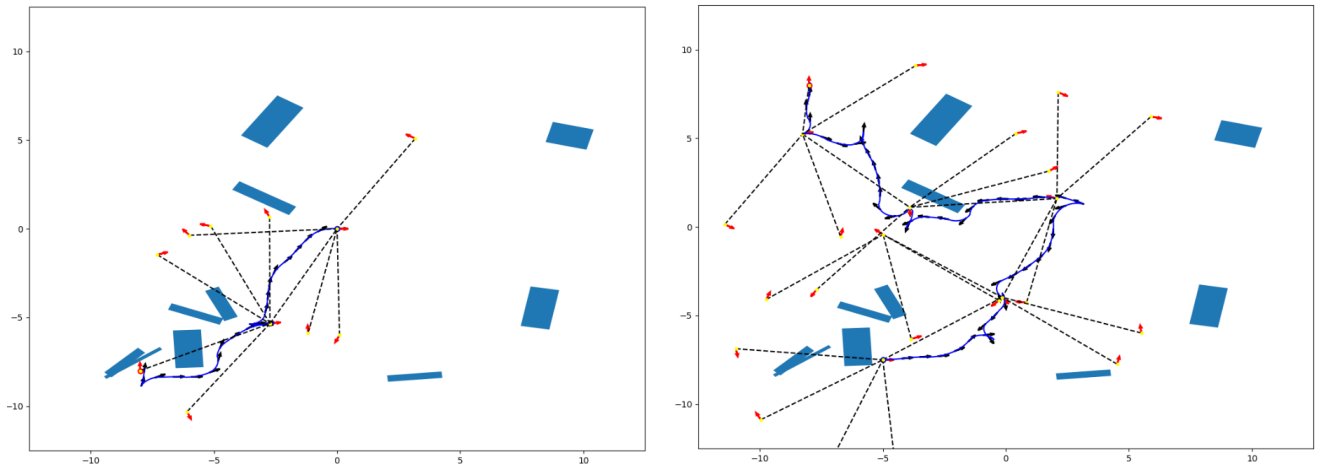

Fig. 9. R* Generated Trajectory with high-level graph

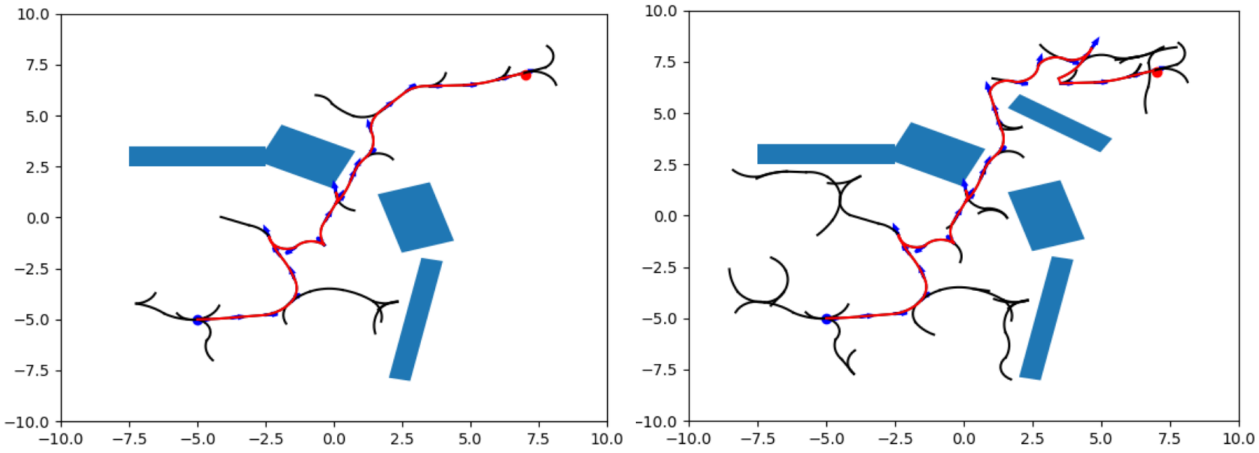

Fig. 10. Original RRT (left) and Updated RRT after encountering new obstacle

how feasible and smooth it was. The results of our evaluation can be found in Table I.

TABLE I
EVALUATION RESULTS ON A STATIC ENVIRONMENT

Trajectory Planner	**Results**		
	Average (m) Pathlength (m)	***Average Duration (s)***	***Trajectory Feasibility***
RRT	11.76	0.62	Bad
A*	6.205	0.56	Good
R*	9.83	3.92	Ok to Bad

Based on the results in Table I, we determined that the A* algorithm provided us with the best combination of trajectory feasibility, trajectory accuracy, and speed of trajectory generation. R* did not provide any benefit over A* for the chosen environment. We conjecture that, with more complicated environments, R* would provide trajectories much faster than A*. However, given the environment that was created for this paper, R* generates a sub-optimal trajectory without providing any noticeable performance in speed. In fact, R*

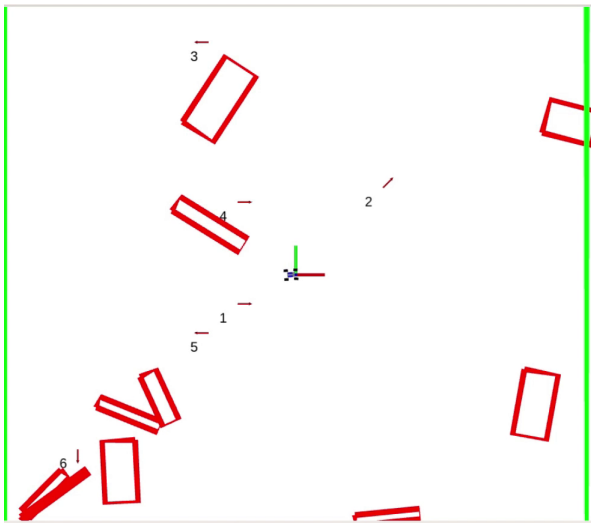

Fig. 11. Evaluation Scenario with Waypoints

takes significantly more time to converge.

V. Extension: Dynamic Environment

In a real-world scenario, the obstacle map is usually not known apriori and as the vehicle moves towards the goal, it encounters new obstacles. With the updated environment, the old trajectory may not be feasible. A naive solution is to re-plan from scratch from the current location to the goal location every time new obstacles are encountered. But we can do better than that.

Considering the scope of the project and time constraint, we only implemented the extension for the RRT algorithm. Whenever a new obstacle is encountered. We quickly divide the existing tree into two kd-trees. Open kd-tree consists of nodes that are reachable from the start location and closed kd-tree consists of nodes from which we can reach the goal. We expand the open kd-tree with standard RRT rules. If a newly added node in the open kd-tree coincides with the closed kd-tree node, then we link those two nodes and now we have obtained the new feasible path. Figure 10 demonstrates how after the addition of a new obstacle new feasible path is obtained.

VI. Conclusion

If we are interested in quickly finding "a" solution without worrying about the length of the path or how many times we have to switch between forward and reverse, then RRT is the way go. But if we are looking for optimum or close to optimum solution in decent time, weighted A* is the way to go. While initially, we expected R* to outperform A* in terms of computation time, In our environment, R* was the slowest and gave okay results. For the bigger environment, R* might converge if hyperparameters are tuned properly. Overall weighted A* seems to work best in our environment both in terms of computation time and quality of the trajectory.

VII. Future Work

The RRT's limitation of not so optimal results is addressed in the RRT* variant that performs the rewiring to get shorter trajectories. It would be interesting to compare RRT* results with the A*, R*.

Handling the change in the environment efficiently is a difficult problem and poses unique challenges in all three methods. More thorough investigations are required in each of them to better handle the change or uncertainty in the environment.

References

[1] LaValle, Steven M. "Rapidly-exploring random trees: A new tool for path planning"

[2] Maxim Likhachev, Anthony Stentz "R* search".

[3] A* search algorithm en.wikipedia.org/wiki/A*_search_algorithm