Final Project Report:

An Implementation of RT-RRT* for Real-Time Path Planning with Dynamic Obstacles

CSC2630: Introduction to Mobile Robotics
University of Toronto
Fall 2022

Charie Brady (1009240563)

Abstract

This project implements a version of RT-RRT* based on the paper "RT-RRT*: A Real-Time Path Planning Algorithm Based On RRT*" by Naderi et al. (2015). The implementation is simulated and tested in a 2D environment with a single dynamic obstacle. The agent plans and moves along a path while avoiding collision with the obstacle. When plans become obstructed, the agent must replan the path in real-time. Based on these tests, it is concluded that while RT-RRT* finds efficient paths comparable to RRT*, this implementation takes significantly more time to generate end-to-end plans due to multiple replanning and rewiring of the tree. Overall, this implementation of RT-RRT* effectively plans paths that avoid a dynamic obstacle within this project's simulated environment.

1 Introduction

Rapidly-exploring random trees (RRTs) are widely used sampling-based path planning algorithms that iteratively expand branches throughout the entire planning space until a viable path to a goal state is found. Variants based on RRT* are guaranteed to find asymptotically optimal paths and have therefore been used for a wide range of applications within robotics (Naderi et al., 2015). RT-RRT* is a variant developed for real-time path planning within environments with dynamic obstacles. This project implements a version of the RT-RRT* algorithm presented in the paper "RT-RRT*: A Real-Time Path Planning Algorithm Based On RRT*" by Naderi et al. and simulates a basic scenario where an agent moves through the sample space towards a target state while avoiding collisions with a moving obstacle. This requires real-time replanning of paths once an initial path becomes obstructed. A video from the author's simulation of RT-RRT* can be viewed here, and a video from this project's simulation of RT-RRT* can be viewed here.

The problem space is a bounded 2D plane containing a subset of static obstacles (e.g., walls or blocks) as well as a single dynamic obstacle moving within the space. In this program, random environments can be generated based on a user-defined number of blocks and walls. There is also a default environment with maze-like static obstacles, which can also be overlayed with randomly generated blocks and walls. The scaling of the environment is also user-defined,

along with a set of other variables, including the initial tree size $\boldsymbol{n}$, maximum k-steps $\boldsymbol{k}$, and collision radius $\boldsymbol{r}$. For simplicity, the program generates a single dynamic obstacle that is restricted to a repetitive motion within a subset of free space. However, multiple dynamic obstacles can be added with minor modification to the program to resemble the authors' simulation. Furthermore, this algorithm allows for multi-query tasks (i.e., path-planning for multiple goal states), and by clicking on the environment, the destination state will immediately update and a path will be planned towards the new destination state, $\mathbf{x}_{\text{target}}$.

The task of RT-RRT* is to find a path from a start state $\mathbf{x}_{\text{start}}$ to $\mathbf{x}_{\text{target}}$ such that the agent travelling the path does not collide with a dynamic obstacle. This requires making real-time path-planning decisions if a collision becomes possible along the initial path. The root of the tree, denoted $\mathbf{x}_0$, changes as the agent moves, and the set of $\boldsymbol{k}$ next planned moves is denoted ($\mathbf{x}_0$, $\mathbf{x}_1$, …, $\mathbf{x}_k$). While at some state $\mathbf{x}_i$ along the initial path, if the agent is expected to be within some radius $\boldsymbol{r}$ of a dynamic obstacle in the following state $\mathbf{x}_{i+1}$, a new path from $\mathbf{x}_i$ to $\mathbf{x}_{\text{target}}$ must be found such that all following states are at least $\boldsymbol{r}$ distance from the obstacle. As Naderi et al. explain, updating the path using a real-time response requires that the next intermediate state, $\mathbf{x}_{i+1}$, be chosen whether or not a path from $\mathbf{x}_{i+1}$ to $\mathbf{x}_{\text{target}}$ has been found. Moreover, the minimal length path should be chosen based on "cost-to-reach" values $\boldsymbol{c}_i$ which are the Euclidean distances from $\mathbf{x}_i$ to $\mathbf{x}_{\text{target}}$ (Naderi et al., 2015). A set of $\boldsymbol{k}$ planned moves make up the new path and the root node is again changed as the agent moves. This project's implementation includes a feature that first searches for paths within the neighborhood of $\mathbf{x}_{\text{target}}$ that are obstacle-free before planning only $\boldsymbol{k}$ steps ahead, which in some scenarios reduces the number of replans needed to avoid a collision. The specifics are detailed in the Methodology section of this paper.

In RT-RRT*, several sampling methods for tree expansion are used based on certain conditions, as detailed in the paper. For this project, I simplify the sampling method by using a uniform distribution only, removing the ellipsoid sampling method of the Informed-RRT* approach. This project's implementation is a simplified version of the RT-RRT* path planner that supports multi-query target states chosen by the user and is simulated using a basic 2D image to illustrate the rewiring and replanning when a dynamic obstacle crosses the agent's initial path. It is possible to simulate multiple dynamic obstacles and/or more complex static environments to see how the planner performs as the tree expands and the path changes to avoid multiple collisions. In this paper, simulated tests are limited to a single dynamic obstacle in an open environment without static obstacles, as discussed in the Evaluation section. This allows a fair comparison of the efficiency between this RT-RRT* implementation and RRT* in similar scenarios. Lastly, limitations and ideas for improvement are discussed in the Limitations section.

3 Related Work

Rapidly-exploring random tree (RRT) and its many variants, such as RRT* and Informed-RRT*, are widely used sampling-based path planning algorithms that will iteratively grow to cover the entire planning space. While RRT does not find asymptotically optimal paths, RRT* was developed by Karaman and Frazzoli in 2011 to guarantee convergence towards the

asymptotically optimal path by rewiring over many iterations when shorter routes are found according to a cost-to-go metric (Katrakazas et al., 2015). These algorithms do not update according to dynamic obstacles; for real-time path planning, variants like CL-RRT and RRT^x have been developed. CL-RRT, developed by Kuawata et al. in 2009, continually re-evaluates and prunes the tree based on changing states, allowing the agent to move around new obstacles (Katrakazas et al., 2015). But as Naderi et al. show, since CL-RRT is constantly pruning the tree and therefore removing paths previously found, it is computational slower to find end-to-end paths compared to RT-RRT*, which retains the entire tree as the agent moves through it (2015). RRT^x, developed by Frazzoli and Otte in 2015, extends RRT* for real-time replanning suitable for dynamic environments, but differs from RT-RRT* in that it does not support multi-query tasks (Naderi et al., 2015).

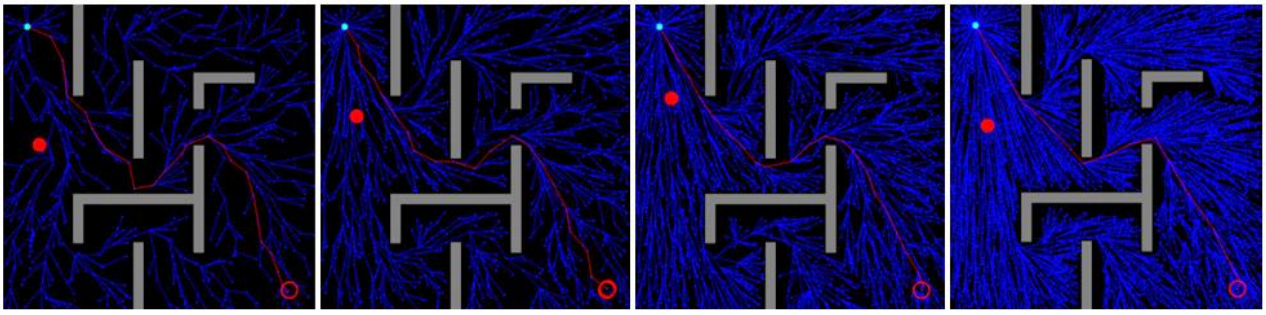
4 Methodology

The RT-RRT* algorithm implemented for this project is identical to Algorithm 1 in the Naderi et al. paper (p. 115, 2015), except that once a path is found, this implementation moves the agent until it reaches the destination rather than for a set time. Also, this implementation uses the full tree, not subtrees $\mathbf{Q}_r$, $\mathbf{Q}_s$.

First, a tree is initialized to a user-defined size **n**, along with initial start and destination states, $\mathbf{x}_{start}$ and $\mathbf{x}_{target}$. Then on each iteration, if the maximum tree size hasn't been reached or on every ***m*** iteration (for random rewiring), the tree is expanded and rewired using Algorithm 2 (p. 115). Then, the destination $\mathbf{x}_{target}$ is updated and a check if a path exists. If no path exists, then the iteration ends and the next iteration begins at tree expansion. If a plan is found, denoted ($\mathbf{x}_0$, $\mathbf{x}_1$, …, $\mathbf{x}_{target}$), then it starts a for-loop that iterates through each step in the plan. At each step $\mathbf{x}_i$ in the plan, collision detection is done on the remaining path ahead ($\mathbf{x}_{i+1}$, $\mathbf{x}_{i+2}$, …, $\mathbf{x}_{target}$). If the path is obstacle free, the agent moves a step along the path; the for-loop ends once the agent reaches the destination. If at some step the path becomes obstructed by an obstacle, then the path needs to be replanned in real-time. The algorithm for replanning is based on Algorithm 6 (p. 117) but with some differences, explained further below. When an obstacle-free replan is found, it will always be to a temporary destination, $\mathbf{x}_{temp}$. The original $\mathbf{x}_{target}$ is saved to memory and the destination is updated to $\mathbf{x}_{temp}$ for the next iteration. Once $\mathbf{x}_{temp}$ is reached, then $\mathbf{x}_{target}$ is popped from memory and the destination is updated to $\mathbf{x}_{target}$ again. In the case that no replan exists, the agent sits and waits until the obstacle moves and a path becomes available. When a destination is reached and memory is empty, the program sits idle aside from tree expansion and random rewiring until a new destination is updated.

Algorithms 2-5 (p. 115-6) are related to tree expansion and rewiring. Algorithm 2 combines tree expansion and rewiring, but in this implementation, I have separated them into two functions as they are used in other areas of the program individually as well as together. For tree expansion, only a simple uniform sampling approach is implemented, unlike the informed-RRT* approach that is incorporated in Naderi et al.'s algorithm. Rewiring is implemented using the classic RRT* approach, where a set of neighbour nodes are reconnected through a center

node based on whether it reduces the total cost to the root. This is done identically to the authors' approach, using Euclidean distance as the cost function. As illustrated in Figure 1, the result of rewiring based on lowest cost neighbors is that as more nodes are added to the tree, paths will approach the asymptotically optimal path. This program utilizes this feature of RRT* to find efficient paths around obstacles. Since sampling is done uniformly, efficient paths throughout all free space can be found quickly in real-time. Algorithm 3 deals with adding nodes to the tree and is implemented identically in this approach. However, one additional step is included: whenever a node is added to the tree, it is checked whether the node is within a user-defined radius from the destination. If so, the shortest available path is found and a plan to $\mathbf{x}_{\text{target}}$ is saved.



n = 500 n = 1000 n = 2000 n = 4000

Figure 1: *An illustration of RRT* with n nodes in the tree. Paths between all pairs of nodes will approach the asymptotically optimal path.*

Algorithms 4 and 5 are implemented differently in this project. Random rewiring is completed in Algorithm 1 at every ***m*** iteration as explained above. Rewiring the root is implemented differently given that the whole tree is processed, not subtrees $\mathbf{Q}_r$, $\mathbf{Q}_s$. Therefore, rewiring the root involves moving the agent along the path by changing the position of the root node in the tree structure. Then, rewiring of the whole tree is only done once the agent reaches the destination or the destination is updated due to replanning. This reduces the processing time of rewiring the root by tracking where the root is as the agent moves, but only rewiring once new plans need to be found.

Lastly, Algorithm 6 (p. 117) deals with replanning when paths become obstructed. The approach is to replan up to ***k*** steps ahead unless an alternative unobstructed path to $\mathbf{x}_{\text{target}}$ is found. This is the same approach taken in this implementation, except that this implementation looks for alternative paths within the neighborhood of $\mathbf{x}_{\text{target}}$ and uses a different method for finding the best k-step path. Algorithm 6 updates the cost for each obstructed node to infinity and rewires the tree. This will cause more paths to rewire around the obstacle. This approach is computationally expensive given that the entire tree needs to be rewired before finding alternative plans. Instead, this implementation generates all unobstructed paths in front of the agent that lead to a node within the neighborhood of the destination—the same neighbors found when rewiring around a node. This is a subset of the tree nodes to consider, and no rewiring is necessary. Of these paths, the 'best' path is chosen by calculating the Euclidean distance between the neighbor and $\mathbf{x}_{\text{target}}$ and taking the minimum, i.e. the path that leads closest to the destination. If a path exists, then replanning is done; the agent moves along the path to the neighbor state,

then rewires the entire tree, then moves from the neighbor to $\mathbf{x}_{target}$. This first strategy for replanning is preferrable to ***k*** step replanning, as it usually requires a single replan and a single rewiring of the tree. This first approach is demonstrated in Figure 2. The red line is the original plan, as it is the shortest path to $\mathbf{x}_{target}$ available. Since the plan is obstructed—at least one node is within the collision radius ***r*** of the obstacle—it requires replanning. By searching for shortest paths to each node within the neighbourhood of $\mathbf{x}_{target}$, an unobstructed plan is found in white.

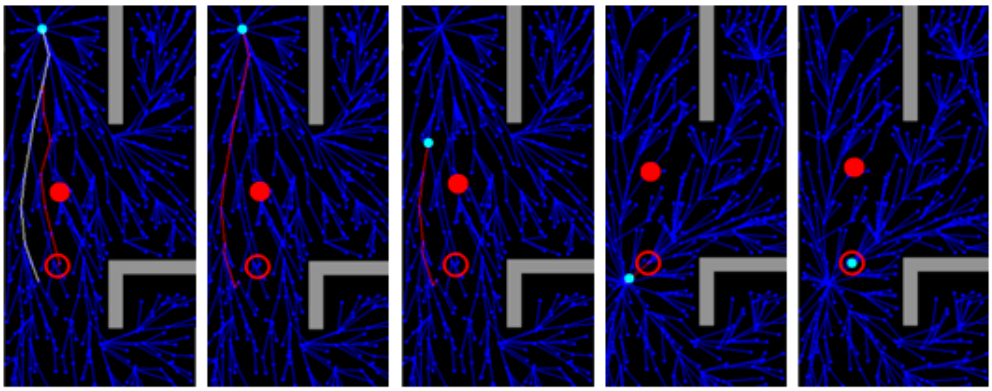


Figure 2: *First replanning strategy: find an obstructed shortest path within the destination's neighborhood.*

The destination is updated to the neighbor and $\mathbf{x}_{target}$ is added to memory. The agent follows the replan until it reaches the neighbor, and only then is the entire tree rewired, as can be seen in the 4th frame, where the neighbor is now the tree root. Then, the original destination is popped from memory and a new path is found to $\mathbf{x}_{target}$. This approach can be faster at replanning paths in real-time in scenarios like the ones simulated in this project, since rewiring the tree is only performed once destinations are reached and new paths are planned.

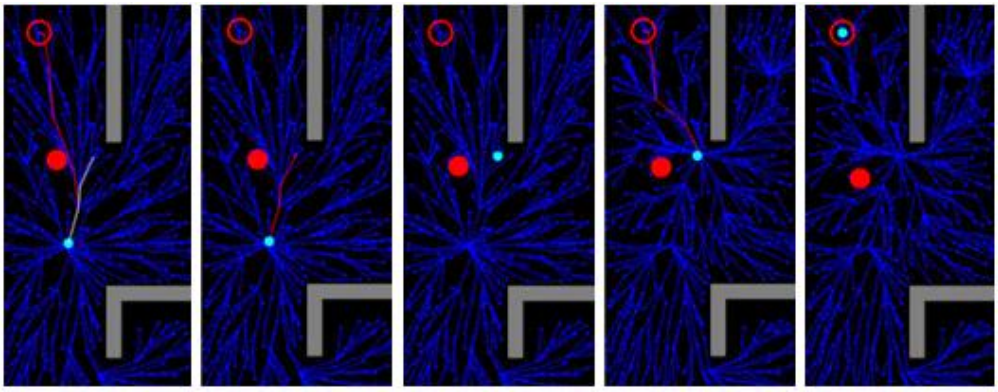


Figure 3: *Second replanning strategy: plan up to k-steps ahead.*

If there are no unobstructed paths within the destination's neighborhood, then we plan up to ***k*** steps ahead by considering all paths to neighbors of the kth node in the original obstructed plan. This will find a temporary destination, $\mathbf{x}_{temp}$. Figure 3 demonstrates this second replanning strategy when an unobstructed path could not be found within the destination's neighborhood.

The original plan is shown in red and the replan ($\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k$) is shown in white. This first replan is found in real-time relatively quickly since it was replanned while at the root, so the entire tree does not have to be rewired as the agent moves around the obstacle. It is only once the agent reaches $\mathbf{x}_{temp}$ that the tree is rewired. However, subsequent replans will be more costly as the entire tree must be rewired at each temporary destination in order to plan the next path. This is illustrated in Figure 4, which illustrates a scenario with multiple k-step replans and rewiring.

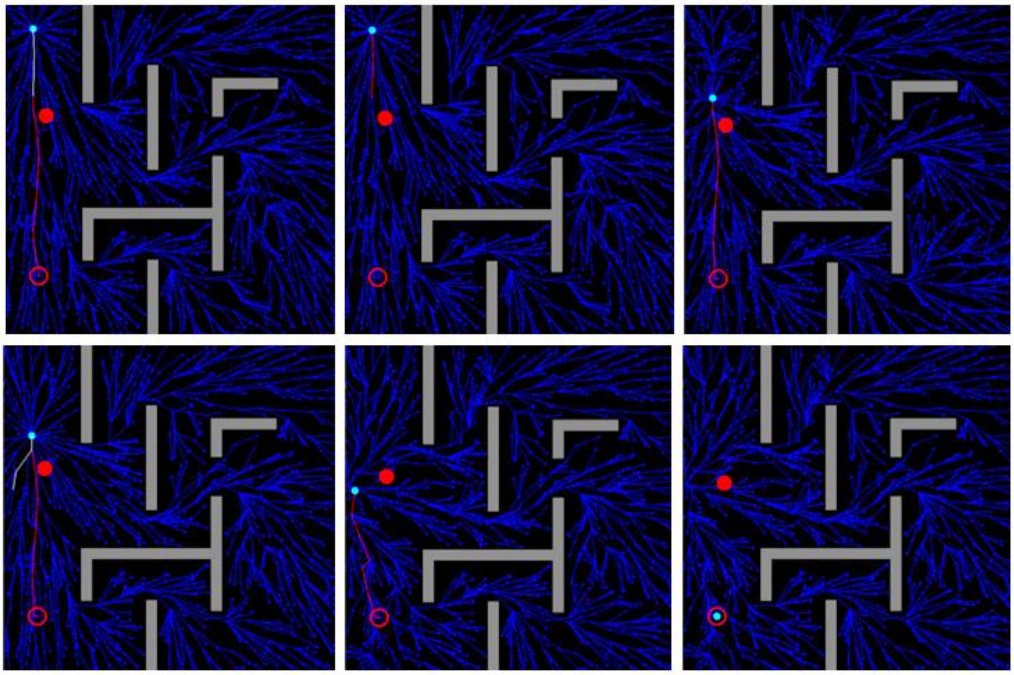


Figure 4: *Multiple k-step replans and subsequent tree rewiring.*

Therefore, the time cost scales with the number of replans required, and so a single replan using the first strategy is preferred. In the uncommon case that no unobstructed paths exist up to $\boldsymbol{k}$ steps ahead, then the agent sits and waits at $\mathbf{x}_0$ until the obstacle moves and a path becomes available, like in Algorithm 6.

5 Evaluation

To evaluate this implementation's efficiency, I compare the average plan length that RRT* finds for a given pair of start and target states with the average plan length that RT-RRT* finds when a single dynamic obstacle causes at least one replanning. These averages are also compared with the shortest possible path, which is the Euclidean distance between the start and target state.

Additionally, I compare the average time the programs take to move the agent from the start state to the destination. This is not only the processing time to generate plans; it is the time for the entire program to run, including the time the agent moves along the path, the time to

rewire the tree during replanning, and the time to display the images on the screen. This is because it is impossible to know how many times a path will need to be replanned without playing out the simulation, which requires the agent move along the path in constant time while the dynamic obstacle moves around it. Some simulations had unlucky timing and positioning of the obstacle in relation to the agent and required multiple replans as the obstacle continually moved towards it and obstructed its paths. Other times, the agent passed the obstacle as it moved in the opposite direction, requiring a single replan to avoid a collision. Taking averages of the entire simulation captures the uncertainty around the innumerable scenarios possible between the two moving objects and the number of replans required. The average time can therefore be interpreted as an estimate of how much longer it may take for the program to generate viable real-time plans start to finish in the presence of an unpredictable dynamic obstacle, compared to generating and moving along a single viable path in the absence of dynamic obstacles using RRT*. In scenarios where the agent is completely obstructed and no viable path exist, the agent stops and waits for the obstacle to pass until collision-free paths become available. These were rare cases with a high proportion of idle time and therefore are not included in the averages.

The data was generated using four pairs of start/target states, each run four times on each program. Also, each program was run using two tree sizes—500 and 1,000 nodes—to compare the efficiency of paths and time processing twice as many nodes. In total, these data are generated from 64 simulations (32 each for RT-RRT* and RRT*). The testing environment has a single dynamic obstacle and no static obstacles for a simpler comparison.

Table 1: *Summary statistics*

Pair	Shortest path	Algorithm	Mean plan length	Mean time (seconds)	Mean replans	Δ length	Δ time
1	351	RRT*	382	0.24	0	15%	430%
		RT-RRT*	440	1.26	1.625		
2	502	RRT*	550	0.34	0	4%	380%
		RT-RRT*	570	1.64	1.625		
3	562	RRT*	616	0.39	0	5%	260%
		RT-RRT*	648	1.41	1.375		
4	562	RRT*	613	0.39	0	13%	410%
		RT-RRT*	691	1.97	1.875		
Total	-	RRT*	540	0.34	0	9%	360%
		RT-RRT*	587	1.57	1.625		

As expected, RRT* finds shorter plans than RT-RRT* for each pair, as seen in Table 1. In total, RT-RRT* plans are 9% longer than RRT* on average but range between 4-15% depending on the pairs tested. RRT* plans are on average 9% longer than the optimal shortest path (MOE ± 2.2%), and RT-RRT* plans are on average 19% longer than the optimal shortest path (MOE ± 4.6%). RT-RRT* plans have higher variability with respect to the optimal path, whereas RRT* plans are more consistently close to optimal. Overall, RT-RRT* finds efficient paths when avoiding dynamic obstacles given that the underlying tree is built using RRT*.

Comparing the average time to reach the destination, RT-RRT* is significantly longer than RRT*, ranging between 260-430% longer depending on the pair. The mean time for RT-RRT* is 1.57 seconds (MOE ± 0.98) whereas RRT* is 0.34 seconds (MOE ± 0.1). Clearly, RT-RRT* has far higher variability (nearly 10x higher) in the time it takes to reach the destination than RRT*. This is because RRT* need only find a single plan, whereas RT-RRT* replans at least 1-2 times on average, but in outlier scenarios may need to replan 3+ times to avoid repeated collisions. Although the paths found will be close to RRT*, a downside of RT-RRT* is that it takes much longer to reach the destination due to collision detection, replanning, and rewiring.

Table 2: *Summary statistics by tree size*

Pair	Shortest path	Algorithm	Nodes	Mean plan length	Mean time (seconds)	Mean replans
1	351	RRT*	500	397	0.19	0
			1000	367	0.29	0
		RT-RRT*	500	446	1.03	2.25
			1000	434	1.5	1
2	502	RRT*	500	584	0.27	0
			1000	515	0.4	0
		RT-RRT*	500	562	0.83	1.5
			1000	577	2.44	1.75
3	562	RRT*	500	638	0.32	0
			1000	595	0.47	0
		RT-RRT*	500	683	0.95	1.5
			1000	612	1.88	1.25
4	562	RRT*	500	633	0.29	0
			1000	593	0.49	0
		RT-RRT*	500	648	1.29	2
			1000	734	2.66	1.75
Total	-	RRT*	500	563	0.26	0
			1000	517	0.41	0
		RT-RRT*	500	585	1.02	1.8125
			1000	589	2.12	1.4375

As Table 2 shows, RRT* plans always become shorter on average when more nodes are added to the tree. However, this is not the case for RT-RRT*. Since there are two methods for replanning (finding a single full replan to a neighbor of the destination, or if one is not available, planning k-steps towards the destination) a larger tree means that RT-RRT* is more likely to find full replanned paths and less likely to use multiple k-step replans to avoid obstacles. These plans towards a neighbor state are longer than paths directly towards the destination, and so as nodes increase, RT-RRT* may find slightly longer paths. This has the benefit of a smaller number of replans on average with larger trees (1.44 vs. 1.81), which decreases replanning time. However, this faster replanning does not offset the slower rewiring step since the program must process twice as many nodes. Overall, the effect is that adding more nodes to the tree slows down RT-RRT* significantly but does not shorten the paths as it does for RRT*. Therefore, smaller trees are more efficient for RT-RRT*, whereas larger trees may be more efficient for RRT*.

6 Limitations

As discussed above, one major limitation with this implementation of RT-RRT* is that it takes significantly longer to generate end-to-end plans compared to RRT* due to repeated replanning and rewiring of the tree. Since rewiring the tree between planning is the most time-consuming task, optimizing the rewiring procedure by storing neighbors in a kd-tree or other more efficient data structure and/or using multithreading to rewire nodes may improve the performance of this algorithm. As Tables 1 and 2 show, larger trees may not improve the average length of paths enough to offset the increased processing time of rewiring a larger tree. Using a smaller base tree along with a path-smoothing overlay might improve the speed of RT-RRT* while offsetting the less efficient and jagged paths produced by a smaller tree.

RT-RRT* is also limited in its ability to prevent collisions. When the agent reaches its destination or no obstacle-free paths exist towards it, the agent sits and waits for either the destination to update or, if it hasn't reached its destination, for a path to become available. While the agent will stop moving forward if it could collide with an obstacle, the obstacle may still collide with the agent while it sits idle. This algorithm only plans collision-free paths towards the target state and will not move to avoid an obstacle colliding into it while stationary. Further, as the authors mentioned in their paper, RT-RRT* only works in bounded environments. Obstacles that approach from outside the bounded space will not factor into the agent's planning, which could result in collisions. Simulating and testing using multiple dynamic obstacles with a robust motion-planning model would be an important next step.

Lastly, while replanning long paths when the agent is relatively distant from the obstacle produced quick and efficient collision-free paths, the availability of alternative paths shrinks when the agent is very near the obstacle, which often results in many small k-step replans. This is time consuming and has the inverse quality of being slowest when closest to a moving obstacle, which is undesirable in real-time path planning. The extent to which the author's addition of Informed-RRT* offsets this feature is unclear, but based on this project's simplified implementation, it is a major limitation of this algorithm.

7 Conclusion

While there exist significant limitations to the viability of this implementation of RT-RRT* in a real-world setting, this algorithm is effective overall at planning paths that avoid a dynamic obstacle within this project's simulated environment. By comparing the average length of paths found with those found using RRT*, it has been shown that RT-RRT* does a good job of finding comparably efficient paths when replanning around an obstacle. Extensions to this project's implementation that address the limitations mentioned in this paper could strengthen the viability of this algorithm for path planning in more complex simulated environments.

References

[1] C. Katrakazas, M. Quddus, W.-H. Chen, and L. Deka, “Real-time motion planning methods for autonomous on-road driving: State-of-the-art and Future Research Directions,” *Transportation Research Part C: Emerging Technologies*, vol. 60, pp. 416–442, 2015.

[2] J. Braun, T. Brito, J. Lima, P. Costa, P. Costa, and A. Nakano, “A comparison of A* and RRT* algorithms with dynamic and real time constraint scenarios for Mobile Robots,” *Proceedings of the 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, 2019.

[3] Otte M, Frazzoli E. RRTX: Asymptotically optimal single-query sampling-based motion planning with quick replanning. The International Journal of Robotics Research. 2016;35(7):797-822.

[4] K. Naderi, J. Rajamäki, and P. Hämäläinen, “RT-RRT*: a real-time path planning algorithm based on RRT*,” *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, 2015.

[5] S. G. Anavatti, M. A. Garratt, and J. Wang, “Real-time path planning algorithm for autonomous vehicles in unknown environments,” *International Journal of Mechatronics and Automation*, vol. 6, no. 1, p. 1, 2017.

[6] 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.

[7] Y. Kuwata, S. Karaman, J. Teo, E. Frazzoli, J. P. How, and G. Fiore, “Real-time motion planning with applications to autonomous urban driving,” *IEEE Transactions on Control Systems Technology*, vol. 17, no. 5, pp. 1105–1118, 2009.