Implementation of RT-RRT* on a point robot

Abhijit Vinod Mahalle
Master of Engineering in Robotics
University of Maryland
College Park, MD, USA
abhimah@umd.edu

Param Ashish Dave
Master of Engineering in Robotics
University of Maryland
College Park, MD, USA
pdave1@umd.edu

***Abstract*—This is a project report on Implementation of RT-RRT* on a point robot. RT-RRT* stands for Real-Time Rapidly Exploring Random Tree. The code for the algorithm was written in Python and the simulation was done using Matplotlib. This project is based on the paper RT-RRT*: A Real-Time Path Planning Algorithm based on RRT* [1].**

I. Introduction

The problem of path planning in dynamic environments is demanding and is encountered in many robotic tasks. Path planning has to be real-time for the robot to react to the changes in the environment like the change in goal or obstacle location. Most of the traditional algorithms like Dijkstra, A*, weighted A*, RRT, RRT*, etc. only perform well in static environment. Algorithms based on real-time path planning should react to the changes in the environment and constantly look for a better path to the goal point. The problem of path planning gets even more challenging in case of multi-query tasks i.e. when the goal-point can be changed. Most of the current approaches of solving the real-time path planning problem falls into three main categories, heuristic methods, potential field methods, and sampling based methods. The real-time version of A* faces the problem of prolonged search time in an environment that has many complex obstacles. Also, A* is a graph-based method and needs discretization of the environment whose resolution has a strong effect on the search time. Real-time implementation of algorithms based on potential field methods suffer from trapping into local minima. Paper[1] proposes the first real-time version of algorithm based on RRT that retains the whole tree in the environment and rewires the the nodes of the tree based on the location of the tree root and changes in the dynamic obstacles. RT-RRT* has been leveraged to find a path with moving goal location in this project.

II. Related Work

A. Real-time Path Planning methods

Search based real-time path planning methods mostly stem from RRTs. Two of these methods are ERRT and CL-RRT. The search tree generated by these algorithms covers only a small portion of the environment, which reduces the search time but increases the length of the path to the goal. At each iteration, ERRT creates a tree using some way-points of the previous iteration. On the other hand, CL-RRT ensures that the agent will not deviate from the planned path and prunes infeasible branches when the agent moves on the tree. Contrary to these methods, RT-RRT* retains the tree between the iterations and changes the tree root when the agent moves. It also rewires the tree when the tree root changes or a dynamic obstacle blocks a node. This makes RT-RRT* to have, fewer iterations to search for a path to various goal points as the tree grows, and the path to those goal paths are shorter compared to CL-RRT. This makes RT-RRT* more suitable for multi-query tasks, i.e. changing goal positions on the fly.

Potential based real-time path planning methods treat the environment as a potential field such the goal point attracts and the obstacles repulse the agent. These methods suffer from the problem of trapping into local minima. This makes them need additional effort to overcome the problem of local minima and to find a path with minimum length to the goal point.

Graph based real-time path planning methods methods usually make a grid of the environment and apply real-time version of A* on it. The disadvantage of graph based real-time path planning method is that even though the environment is explored and a graph representing it is constructed, one needs further processing to extract the path from the graph. In multi-query tasks, graph based path planning methods may need to search the entire graph to find a path to different goal points. Unlike this, RT-RRT* needs fewer iterations for finding goal points as the tree grows in the environment. Besides, it utilizes the efficient tree structure to return a path to possibly multiple goal points by backtracking the ancestors from the goal.

III. Notation

IV. Method

The RT-RRT* algorithm was implemented on a point robot with changing goal position. Since the algorithm is for real-time robot, the point robot moves one step along the path in each iteration. If in a given iteration, the path to the goal is not found, path is found to the node closest to the goal node by backtracking on the search tree. The RT-RRT* algorithm has five sub-algorithms to find a path to the goal in each iteration. Each of the sub-algorithms are explained in great detail below.

The task is to find a near-optimal path from the robot to the goal where x_{goal} can be changed on the fly. To

Algorithm 1 RT-RRT*: Our Real-Time Path Planning

1: **Input:** $\mathrm{x_a}$, $\mathcal{X}_{\mathrm{obs}}$, $\mathrm{x_{goal}}$
2: Initialize $\mathcal{T}$ with $\mathrm{x_a}$, $\mathcal{Q}_r$, $\mathcal{Q}_s$
3: **loop**
4: `Update` $\mathrm{x_{goal}}$, $\mathrm{x_a}$, $\mathcal{X}_{\mathrm{free}}$ and $\mathcal{X}_{\mathrm{obs}}$
5: **while** time is left for `Expansion` and `Rewiring` **do**
6: `Expand` and `Rewire` $\mathcal{T}$ using Algorithm 2
7: Plan $(\mathrm{x_0}, \mathrm{x_1}, ..., \mathrm{x_k})$ to the goal using Algorithm 6
8: **if** $\mathrm{x_a}$ is close to $\mathrm{x_0}$ **then**
9: $\mathrm{x_0} \leftarrow \mathrm{x_1}$
10: `Move` the agent toward $\mathrm{x_0}$ for a limited time
11: **end loop**

Fig. 1. RT-RRT*

find a near-length path (minimal length path), we had used cost-to-come(c_i) which is computed using the Euclidean length of the path from x_0 to x_i. Since the algorithm is to be deployed on real-time, there is a limited amount of time for Tree Expansion and Rewiring and path planning. When the time for path planning is up, next immediate node after x_0 in the planned path $(x_0,\ x_1,\ ...,\ x_k)$, x_1, is committed and should be followed. In RT-RRT*, x_0 is changed when the agent moves to keep the agent near the tree root, and by retaining the whole tree between iterations a path to any point in the environment can be returned. Therefore, the challenge is how to rewire the tree really fast and in a real-time manner to react to the changes in the environment (changes in X_{obs}) as well as to return a minimal length path to x_{goal} while x_{goal} can be any point in the environment.

A. RT-RRT*

The RT-RRT* is introduced in Algorithm 1 and it interleaves path planning with tree expansion and rewiring. In the first iteration, the root of the tree x_0 is initialized at x_a (line 2). The loop runs until a path is found between the robot position and goal position. At each iteration, the tree is rewired and expanded for 0.5 seconds (lines 5-6). The path is planned for from the current tree root five steps further (k in line 7). The planned path is a set of nodes starting from the tree root, $(x_0,\ x_1,\ ...,\ x_k)$. At each iteration, the point robot is moved one step in the planned path along with the tree root, x_0 (lines 8-9). This enables the robot to move towards the goal before getting the complete path and it can move on a real-time. Line 4 of algorithm 1 updates the goal point, position of the agent and the obstacles that is used later in the Path planning and the Tree Expansion-and-Rewiring algorithms.

B. Tree Expansion and Rewiring

Tree Expansion-and-Rewiring is introduced in Algorithm 2. A random node $(x_{rand}$ is generated in each iteration (line 2). The node that is at a lowest Euclidean distance from the sampled node is considered as $(x_{closest}$. All the nodes in the

Algorithm 2 Tree Expansion-and-Rewiring

1: **Input:** $\mathcal{T}$, $\mathcal{Q}_{\mathrm{r}}$, $\mathcal{Q}_{\mathrm{s}}$, k_{max}, r_s
2: `Sample` $\mathrm{x_{rand}}$ using (1)
3: $\mathrm{x_{closest}} = \arg\min_{\mathrm{x} \in \mathcal{X}_{\mathrm{SI}}} \mathtt{dist}(\mathrm{x}, \mathrm{x_{rand}})$
4: **if** $\mathtt{line}(\mathrm{x_{closest}}, \mathrm{x_{rand}}) \subset \mathcal{X}_{\mathrm{free}}$ **then**
5: $\mathcal{X}_{\mathrm{near}} = \mathtt{FindNodesNear}(\mathrm{x_{rand}}, \mathcal{X}_{\mathrm{SI}})$
6: **if** $|\mathcal{X}_{\mathrm{near}}| < k_{\mathrm{max}}$ or $|\mathrm{x_{closest}} - \mathrm{x_{rand}}| > r_s$ **then**
7: $\mathtt{AddNodeToTree}(\mathcal{T}, \mathrm{x_{rand}}, \mathrm{x_{closest}}, \mathcal{X}_{\mathrm{near}})$
8: `Push` $\mathrm{x_{rand}}$ to the first of $\mathcal{Q}_{\mathrm{r}}$
9: **else**
10: `Push` $\mathrm{x_{closest}}$ to the first of $\mathcal{Q}_{\mathrm{r}}$
11: $\mathtt{RewireRandomNode}(\mathcal{Q}_{\mathrm{r}}, \mathcal{T})$
12: $\mathtt{RewireFromRoot}(\mathcal{Q}_{\mathrm{s}}, \mathcal{T})$

Fig. 2. Tree Expansion-and-Rewiring

$$\mathrm{x_{rand}} = \begin{cases} \mathrm{LineTo}(\mathrm{x_{goal}}) & \text{if } P_r > 1-\alpha \\ \mathrm{Uniform}(\mathcal{X}) & \text{if } \begin{cases} P_r \leq \dfrac{1-\alpha}{\beta} & \text{or} \\ \nexists \mathrm{path}(\mathrm{x_0}, \mathrm{x_{goal}}) \end{cases} \\ \mathrm{Ellipsis}(\mathrm{x_0}, \mathrm{x_{goal}}) & \text{otherwise} \end{cases}$$

line between the sampled node and the closest node should not be in the obstacle space. If any of the node is in the obstacle space, the sampled node is discarded. If not, all the nodes near to $(x_{rand}$ within a certain threshold are found and added to the list $(x_{near}$. If the number of nearby nodes id below $(k_{max}$ or the distance between the sampled and its closest node is above $(r_s$ then the node is added to the tree, the algorithm for which is given in algorithm 3. The condition in line 6 is used to control the density of the generated nodes on the tree. After this, the random nodes and its nearby nodes are rewired as per algorithm 4 and the root node is rewired as per algorithm 5 that are explained in great detail below. The sampled node $(x_{rand}$ is always used in rewiring random parts of the tree either around itself or around its closest node, $x_{closest}$ (lines 8, 10, 11).

C. Random sampling

Random sampling in line 2 of algorithm 2 uses the following equations. $(P_r$ is a random number between [0, 1] and α is a small user-given constant, e.g. 0.1. $\beta \in \mathrm{R}$ is used to divide the samples between Uniform(X) and Ellipsis $(x_0, x_{goal}$ samplings. When the value of $(P_r$ satisfies the first condition, the random node is generated in such a way that it lies between the line connecting the closest node to the goal node and the goal node. The closest node to the goal node is determined by the Euclidean distance. Uniform(X) samples the environment uniformly. Ellipsis samples inside an ellipsis so that the path

Algorithm 3 Add Node To Tree

1: **Input:** $\mathcal{T}, \mathbf{x}_{\text{new}}, \mathbf{x}_{\text{closest}}, \mathcal{X}_{\text{near}}$
2: $\mathbf{x}_{\text{min}} = \mathbf{x}_{\text{closest}}$, $c_{\text{min}} = \texttt{cost}(\mathbf{x}_{\text{closest}}) + \texttt{dist}(\mathbf{x}_{\text{closest}}, \mathbf{x}_{\text{new}})$
3: **for** $\mathbf{x}_{\text{near}} \in \mathcal{X}_{\text{near}}$ **do**
4: $\quad c_{\text{new}} = \texttt{cost}(\mathbf{x}_{\text{near}}) + \texttt{dist}(\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{new}})$
5: $\quad$ **if** $c_{\text{new}} < c_{\text{min}}$ and $\texttt{line}(\mathbf{x}_{\text{near}}, \mathbf{x}_{\text{new}}) \in \mathcal{X}_{\text{free}}$ **then**
6: $\quad\quad c_{\text{min}} = c_{\text{new}}$, $\mathbf{x}_{\text{min}} = \mathbf{x}_{\text{near}}$
7: $V_{\mathcal{T}} \leftarrow V_{\mathcal{T}} \cup \{\mathbf{x}_{\text{new}}\}$, $E_{\mathcal{T}} \leftarrow E_{\mathcal{T}} \cup \{\mathbf{x}_{\text{min}}, \mathbf{x}_{\text{new}}\}$

Fig. 3. Algorithm of Adding a node to tree

from x_0 to x_{goal} is inside it. x_0 and x_{goal} are focal points of the ellipsis where its transverse and conjugate diameters equal to c_{best} and $\sqrt{c_{best}^2 - c_{min}^2}$. c_{best} is the length of the path from x_0 to x_{goal} and c_{min} is the Euclidean distance between x_0 and x_{goal}.

D. Adding a node to the Tree

As shown in figure, if we want to add a new node x_{new} to the tree in a similar way we add in RRT* algorithm, we find the parent with the minimum cost to go. Cost to go is the euclidean distance of the node and the goal node. We find the parent with minimum cost to go value inside the X_{near}. Now here the calculations for the c_i is proportional to the length between the path x_0 and x_i. Thus the cost(x_i) need to compute the c_0 whenever c_j for any intermediate node in the path to x_i is changed. Therefore, when the costx_i is called, the algorithm will redo all the computation related to the c_i starting from x_i and ending on x_0 only if the condition that a new node had been added to the path to x_i or any new change in the node x_0 has happened. Apart from this the other condition is if there is any change in x_0 or dynamic obstacle is inside r_0. Whenever we access the cost(x_i , if one of the ancestors of x_i is blocked by a dynamic obstacle node, where the cost c_i is infinity, we will block that node as well. This cost(x_i) is used in further sub-algorithms. The nodes and edges in the tree are denoted by V_T and E_T. When a node is added to the tree dictionary, it makes the tree bigger i.e it expands the tree. Now if it is inside the goal as shown in algorithm, a new path to x_0 is found and we can update the neighbouring grid squares used for grid based indexing which is spatial in nature. the algorithm is using a tree structure to build the tree and each node ion that tree has easy access to its child and its parent. Thus we can conclude the we may use E_T in the explanation only.

E. Tree Largeness

The tree grows too large by retaining T between iterations. For the sake of simplicity and saving memory grid-based spatial indexing (Fig 4), however one might use KD-Tree spatial indexing to gain even more speed up. X_{SI} is used to find $x_{closest}$ and X_{near} of x_{rand} in Algorithm 2. Also, X_{SI} is used in Algorithms 4 and 5 to search for X_{near} around x_r and x_s, respectively. To find X_{SI} around one node (e.g. x_u,), we find the grid square g_u,, where x_u, is located. Then, we return all nodes of the tree that are inside u and in its adjacent

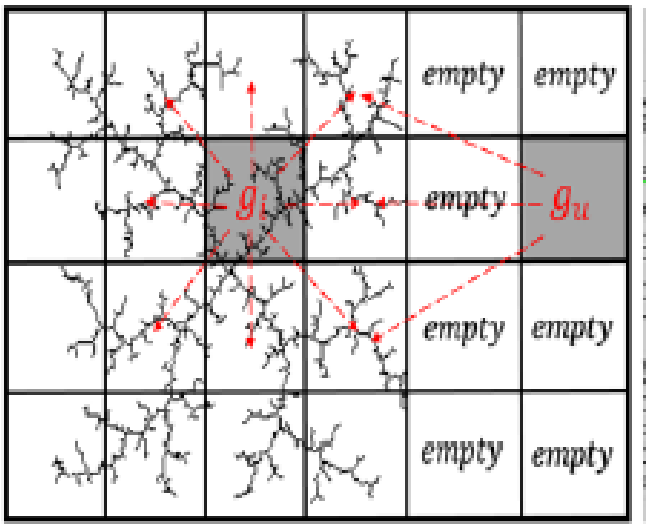

Fig. 4. Adjacent grid

squares. A grid square gj is considered adjacent to other grid square gi when gj is the closest grid square to gi which has at least one node of the tree inside it. Fig 2 illustrates the difference between adjacent grid squares for gi and u. Note that, the size of the grid affects strongly the processing time and the size of X_{SI}. Also, each grid should be big enough to contain the obstacle region around each dynamic obstacle (see Section 5 for our obstacle and grid details).

F. Rewiring the Tree

As seen in the figure, we rewire the node x_i when it gets a lowest cost to come c_i value by passing from another new node instead of its old parent node. Thus in our algorithm written the rewiring should be only done around the new node in focus. Rewiring basically means we change the parent of the new node and make the node in focus as the parent of the new node given the new node has low cost to go. Also rewiring of nodes should be done around the new node that is being added similar to that of RRT* as wella se one where one has already been added around the node which is opposed to the RRT* due to changing the tree root and obstacles which are dynamic in nature. When one of the x_0 or any dynamic obstacle is in the r_0 changes its stare, we will be required to require a large portion of the tree which is done as follows. First we rewire a random part of the tree as per our algorithm 4 where we pop a element out of our queue and check foe its near nodes. When the new cost to go of the near node is less than previous cost, we make change to the tree and male that node as parent.

The lines from three to seven in algorithm 4 and similarly lines five and nine in algorithm 5 will help in rewiring the neighbor node. X_{near} if nodes x_r and x_s. The rewiring of the tree is done in vicinity of x_{near} if and only if by making alterations to the path which is current parent to the node x_j ; c_i for x_{near} reduces where ever x_j is equal to the value

Algorithm 4 Rewire Random Nodes

Input: $\mathcal{Q}_r, \mathcal{T}$
repeat
  $\mathbf{x}_r$=PopFirst($\mathcal{Q}_r$), $\mathcal{X}_{near}$=FindNodesNear($\mathbf{x}_r, \mathcal{X}_{SI}$)
  for $\mathbf{x}_{near} \in \mathcal{X}_{near}$ **do**
    c_{old}=cost($\mathbf{x}_{near}$), c_{new}=cost($\mathbf{x}_r$)+dist($\mathbf{x}_r, \mathbf{x}_{near}$)
    if $c_{new} < c_{old}$ and line($\mathbf{x}_r, \mathbf{x}_{near}$) $\in \mathcal{X}_{free}$ **then**
      $E_\mathcal{T} \leftarrow (E_\mathcal{T} \setminus \{\text{Parent}(\mathbf{x}_{near}), \mathbf{x}_{near}\}) \cup \{\mathbf{x}_r, \mathbf{x}_{near}\}$
      Push $\mathbf{x}_{near}$ to the end of $\mathcal{Q}_r$
until Time is up or $\mathcal{Q}_r$ is empty.

Algorithm 5 Rewire From the Tree Root

Input: $\mathcal{Q}_s, \mathcal{T}$
if $\mathcal{Q}_s$ is empty **then**
  Push $\mathbf{x}_0$ to $\mathcal{Q}_s$
repeat
  $\mathbf{x}_s$=PopFirst($\mathcal{Q}_s$), $\mathcal{X}_{near}$=FindNodesNear($\mathbf{x}_s, \mathcal{X}_{SI}$)
  for $\mathbf{x}_{near} \in \mathcal{X}_{near}$ **do**
    c_{old}=cost($\mathbf{x}_{near}$), c_{new}=cost($\mathbf{x}_s$)+dist($\mathbf{x}_s, \mathbf{x}_{near}$)
    if $c_{new} < c_{old}$ and line($\mathbf{x}_s, \mathbf{x}_{near}$) $\in \mathcal{X}_{free}$ **then**
      $E_\mathcal{T} \leftarrow (E_\mathcal{T} \setminus \{\text{Parent}(\mathbf{x}_{near}), \mathbf{x}_{near}\}) \cup \{\mathbf{x}_s, \mathbf{x}_{near}\}$
    if $\mathbf{x}_{near}$ is not pushed to $\mathcal{Q}_s$ after restarting $\mathcal{Q}_s$ **then**
      Push $\mathbf{x}_{near}$ to the end of $\mathcal{Q}_s$
until Time is up or $\mathcal{Q}_s$ is empty.

Fig. 5. Algorithm of Rewiring the tree

of nodes x_r and x_s The main difference between all the algorithms 4 and 5 is that there is a different focus point of rewiring. The algorithm 5 rewires random part of thr tree starting from our node x_i around a random node x_{rand} or a closest node $x_{closest}$ that are added to the deque queue Q_r in already explained algorithm 2. Now if any rewiring is to occur to any of the near node x_{new} , the algorithm adds that node x_{near} to the deque queue Q_r since node around that node have the chance to be a potential rewire candidate.On other hand, the next algorithm rewire form root node, focuses rewiring the entire tree around the root node x_0 and thus around the agent node x_i.

Algorithm 5 focuses on rewiring node x_0 and push all its nearing neighbor nodes to Q_s. Then it will again recursively push new node with distance greater than x_0 bu queue,pop nodes x_i from the queue Q_s amd pushing the x_{near} around the periphery of x_s when the condition is met that is x_i is not pushed when restarting. Using the Q_s allows us to rewire the near nodes with the same cost to go values from the root node. We would refer the nodes with the same cost to go as rewiring radius circle from x_0. We rewire at each iteration until Q_i is empty or the time stops. if time is up and Q_i is not empty , we rewire the nodes in upcoming iterations.

V. Simulation

As the algorithm is complex, it is was not possible to simulate the robot using ROS in gazebo.So as mentioned in the project proposal, we implemented our fallback plan. We simulated our algorithm on a point robot using python as our script as it is easy to prototype in python. We used a 10x10 grid as our World Map. We used two rectangles and two circles in the map as shown in figure below. The spawn of the robot is at (-4.0 , -0.5) while the dynamic obstacle moves starting from (3.0 , -0.5) . The dynamic obstacle moves in a direction until it meets an obstacle. When it collides with a obstacle, it spawns in a mirror opposite direction to the current position with respect to y axis. To simulate the algorithm there is a time constraint under which the point robot must find the obstacle. The total time to run the code as according to the algorithm we set as 30 seconds. For each rewiring it takes 0.5 seconds. At each time step it generates 5 lines/Nodes. The other parameters taken into considerations were expand = 0.3 and $rewire_radius$ = 1.2. Here expand is the distance of expansion at each iteration. Rewiring radius is the radius in which rewiring happens at each node. These hyper-parameters determine the efficiency of the code and time it required to reach the goal. As we tune these hyper-parameters, we get different results. As the code uses random exploring trees as core concept, for each test case with same hyper-parameters, we get different explored nodes. But as the library random uses particular Probability Density Function, we can expect some sort of convergence in results leading to proper tuning of parameters. The following figures represent the two iterations of the code. The green diamond is the point robot and the red circle is the dynamic moving obstacle. As observed due to randomness of the exploration of algorithm nodes we may see some randomness but due to presence of cost function and rewiring of nodes we get the same result. We used matplotlib as our medium of output to simulate the output.

VI. Conclusion

In this report we discussed ,analysed and implemented the RT-RRT*(Real-Time Rapidly-Exploring Random Trees) algorithm. As claimed by the authors, it is the first real-time version of RRT* and informed RRT*. RRT algorithm and RRT* algorithms have some draw back as mentioned in the introduction. The algorithm implements real-time capabilities by interleaving the path planning with sub-algorithms of rewiring and tree expansion. One of the highlights of the algorithms is the fact that we move the root of tree with our point robot in order to retain the tre instead of building one new tree every time a new iteration is computed. As per the algorithm the tree must continue to grow until it cover the entirety of the environment. The tree grows while there is a unexplored space in the world map. As the RRT* has some problem converging in a very large environment , the algorithm improves it by introducing two modes of rewiring the nodes. This improvement is for having shorter paths with limited number of nodes in a very large tree. First method is rewiring at the start of the root as explained in algorithm 4. This method creates a growing a circle centered around the agent which keeps growing. This circle rewires each node around the point of focus and thus the tree root. The second method is the rewiring of the random part of the tree. As explained in the algorithm 5, the rewiring is done using both uniform sampling and focused sampling but instead of one node,in a single patch of radius r. Thus as simulation shows,the combination of the two rewiring methods as explained enables the tree to retain its structure and enables

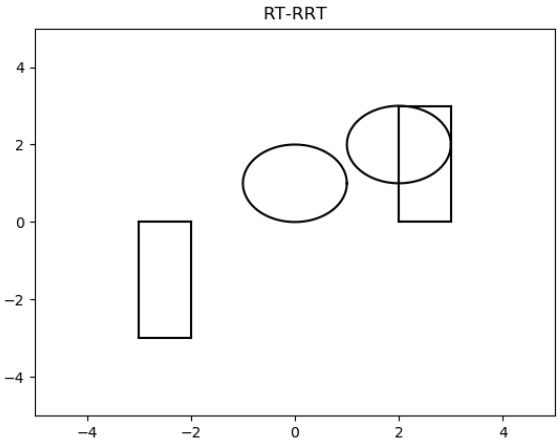

Fig. 6. Obstacle Map

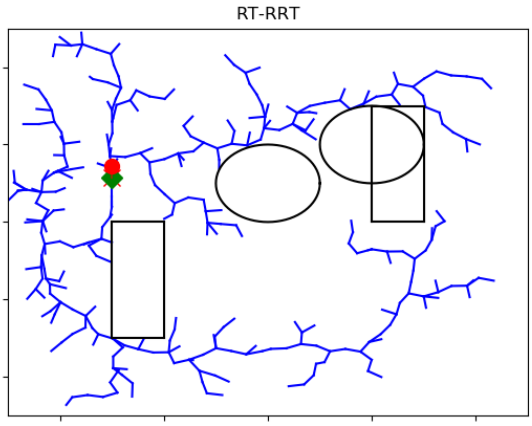

Fig. 8. Simulation Result 2

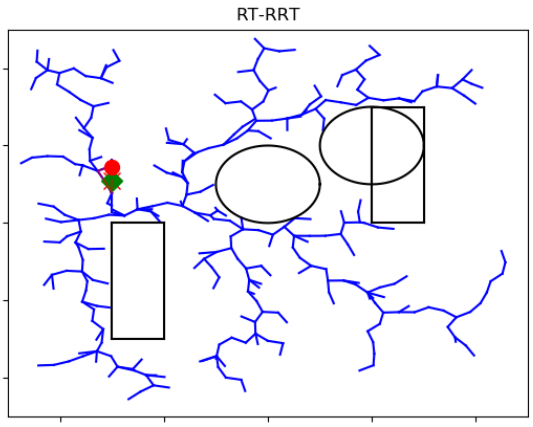

Fig. 7. Simulation Result 1

it to find shorter path with far less iterations for one or multiple goal points. One should note that, for the sake of complexity ,simplicity, memory and time constraint, we used a grid of 10x10 to speed up our neighboring node search using a dictionary. Some more sophisticated spatial index method such as a KD-tree could have been used which would speed up the search even more and allow using bigger sampling budgets. Coming to limitations, Rt-RRT*has its limitations which include memory constraints. The algorithm requires a very large memory capacity because we stote the entire tree in the memory at all times. As we have tested the algorithm in a limited environment, it only works in a bounded environment. Thus, the challenges of unbounded and large distance environments remain to be addressed by the authors and the code. We have made assumptions about objects as obstacles and resorted to circumventing the obstacles with a given minimum distance and tolerance. Further work could include extracting an obstacle model a fruitful research direction for the future.

From the graphs we can see that even though the robot catches upto the dynamic obstacle

ACKNOWLEDGMENT

We are thankful to the Instructor Dr.Reza Monfaredi for guiding us throught the planning course to enable us to understand the paper material and implement it as per required. We are also thankful to the teaching assistants and the grader who helped whenever we were stuck with a difficult concept.

REFERENCES

[1] RASTGOO, M. N., NAKISA, B., NASRUDIN, M. F., NAZRI, A., AND ZAKREE, M. 2014. A critical evaluation of literature on robot path planning in dynamic environment. Journal of Theoretical Applied Information Technology.

[2] SONG, S., LIU, W., WEI, R., XING, W., AND REN, C. 2014. Path planning directed motion control of virtual humans in complex environments. Journal of Visual Languages Computing.

[3] STURTEVANT, N. R., BULITKO, V., AND BJORNSSON ¨ , Y. 2010. On learning in agent-centered search. In AAMAS. SUD, A., ANDERSEN, E., CURTIS, S., LIN, M., AND MANOCHA, D. 2008. Real-time path planning for virtual agents in dynamic environments. In ACM SIGGRAPH 2008 Classes, ACM.

[4] KARAMAN, S., AND FRAZZOLI, E. 2011. Sampling-based Algorithms for Optimal Motion Planning. Int.

[5] J. Rob. Res.. KAVRAKI, L. E., S ˇ VESTKA, P., LATOMBE, J.-C., AND OVERMARS, M. H. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Trans. Robot. Autom..

[6] KHATIB, O. 1986. Real-time obstacle avoidance for manipulators and mobile robots. The International Journal of Robotics Research.

[7] LAVALLE, S. M. 1998. Rapidly-Exploring Random Trees: A new Tool for Path Planning.

[8] CANNON, J., ROSE, K., AND RUML, W. 2012. Real-Time Motion Planning with Dynamic Obstacles. In Symposium on Combinatorial Search.

[9] GAMMELL, D, J., SRINIVASA, S, S., BARFOOT, AND D, T. 2014. Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic. In IROS, IEEE.

[10] GUTMANN, J.-S., FUKUCHI, M., AND FUJITA, M. 2005. Realtime path planning for humanoid robot navigation. In IJCAI.