Dynamic motion planning for robots

Nishad Kulkarni
A. James Clark, School of Engineering
University of Maryland, College Park
UID - 117555431

Yashas Shetty
A. James Clark, School of Engineering
University of Maryland, College Park
UID - 119376348

I. Introduction

A. Definition

Robotic navigation among dynamic obstacles is a very challenging problem. In real time cases the path planning of a robotic agent needs to be dynamic such that the obstacle space and the

B. Background

Real time path planning algorithm struggle with the feasibility of a path versus the fastest path. Many of the current approaches for solving the real-time path planning problem fall into the categories of heuristic methods [1], potential field methods [2] and sampling methods like rapidly exploring random trees (RRTs) [3]. Real time path planning methods are generally of two types,

Tree Based Path Planning methods mostly stem from RRTs.Two of these methods are ERRT [4] and CLRRT [5].

Graph Based Path Planning methods usually make a grid of the environment and apply real-time versions of A* to it. Some of the methods simply divide the environment into simple polygonal grids [6; 1] and some methods that use Voronoi diagram [7] or sampling [8].

C. literature review

Here are some papers we reviewed and considered implementing.

1) **RT-RRT*: A Real-Time Path Planning Algorithm Based On RRT*** link1 - In this paper the authors implement a real time RRT* based method that is time-bound to make decision while moving in a known envionment.
2) **Socially Aware Motion Planning with Deep Reinforcement Learning** link2 - In this work the authors take into account the movement model of humans and their unreliability. The authors focus on what not to do and use Reinforcement learning to develop a decision making policy.
3) **Motion Planning Among Dynamic, Decision-Making Agents with Deep Reinforcement Learning** link3 - Similar to the previous paper, this paper uses Rl to generate a decision making policy as well. However, in this case the authors use LSTMs to generate the policy from a handful of observations.
4) **Path Planning using Neural A* Search** link4 - In this paper, the authors reformulate the A* algorithm to be differentiable and couple it with a convolutional encoder to form an end-to-end trainable neural network planner. This is a data driven approach.

We finally decided to go with paper 1.

II. Method

A. Path Planning method

As of now, the most suitable paper we found for our project uses the RT-RRT* method for path planning. The paper implements the method to a point robot. RT-RRT* stands for RealTime-Rapidly Exploring Random Tree. Please note that depending on feasibility we may even consider other methods like A* depending on the source literature as we are also considering some Reinforcement Learning based collision avoidance methods.

The algorithm is divided into 6 parts. The first part is like a main function which calls the other parts.

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

Input: $\mathbf{x}_a$, $\mathcal{X}_{obs}$, $\mathbf{x}_{goal}$
Initialize $\mathcal{T}$ with $\mathbf{x}_a$, $\mathcal{Q}_r$, $\mathcal{Q}_s$
loop
  `Update` $\mathbf{x}_{goal}$, $\mathbf{x}_a$, $\mathcal{X}_{free}$ and $\mathcal{X}_{obs}$
  while time is left for `Expansion` and `Rewiring` **do**
    `Expand` and `Rewire` $\mathcal{T}$ using Algorithm 2
  Plan $(\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_k)$ to the goal using Algorithm 6
  if $\mathbf{x}_a$ is close to $\mathbf{x}_0$ **then**
    $\mathbf{x}_0 \leftarrow \mathbf{x}_1$
  `Move` the agent toward $\mathbf{x}_0$ for a limited time
end loop

In this, We get inputs of the initial obstacles, goal and current position of the robot, We then initialize the RRT with the current node as its root node of the tree. As the planning is real time, we run a loop till we reach the obstacle or until we cross a specific time limit. In each iteration we update the obstacles,goal (which are dynamic) and the tree. Also the root of the tree can change after every iteration. In each iteration we expand and rewire the tree. Then, just like in RRT*, if we find the goal in the tree we will plan our path to the goal from the root. This plan will be stored in the Q_r queue. Then, if the first node of the plan i.e. x_0 is close to the root node, we change the x_0 to x_1 , in order to optimize the path. Then we move the robot towards x_0 . As this process runs in a while loop, we either reach the goal or cross our time limit.

The second part of the algorithm deals with the tree expansion and rewiring.

Algorithm 2 Tree Expansion-and-Rewiring

Input: $\mathcal{T}, \mathcal{Q}_r, \mathcal{Q}_s, k_{\max}, r_s$
Sample $\mathbf{x}_{\text{rand}}$ using (1)
$\mathbf{x}_{\text{closest}} = \arg\min_{\mathbf{x}\in\mathcal{X}_{SI}} \texttt{dist}(\mathbf{x}, \mathbf{x}_{\text{rand}})$
if $\texttt{line}(\mathbf{x}_{\text{closest}}, \mathbf{x}_{\text{rand}}) \subset \mathcal{X}_{\text{free}}$ **then**
  $\mathcal{X}_{\text{near}} = \texttt{FindNodesNear}(\mathbf{x}_{\text{rand}}, \mathcal{X}_{SI})$
  if $|\mathcal{X}_{\text{near}}| < k_{\max}$ or $|\mathbf{x}_{\text{closest}} - \mathbf{x}_{\text{rand}}| > r_s$ **then**
    $\texttt{AddNodeToTree}(\mathcal{T}, \mathbf{x}_{\text{rand}}, \mathbf{x}_{\text{closest}}, \mathcal{X}_{\text{near}})$
    Push $\mathbf{x}_{\text{rand}}$ to the first of $\mathcal{Q}_r$
  else
    Push $\mathbf{x}_{\text{closest}}$ to the first of $\mathcal{Q}_r$
  $\texttt{RewireRandomNode}(\mathcal{Q}_r, \mathcal{T})$
$\texttt{RewireFromRoot}(\mathcal{Q}_s, \mathcal{T})$

We give the tree, two queues, (root and random), the kmax limit, and radius rs as the inputs. Just like RRT, we sample random points, dded to the tree until it completely covers the environment. And it is rewired to the tree around itself or to the closest point($x_{closest}$) in its surroundings around the tree. While adding nodes to the tree, we make sure of two things, firstly, the number of nodes in the neighbourhood of the tree do not exceed a certain number (which is k_{max}) and secondly, we make sure that the new node is not very close to the $x_{closest}$ (closeness is decided by the radius rs). We added the new node to the Q_r queue. If neither of the above are not satisiied, we add $x_{closest}$ to the Q_s queue. After adding the node, we rewire the tree, based on which queue the node is added. i.e. if the node is added to Q_s , we rewire Q_s to the tree and if the node is added to Q_r , we rewire Q_r to the tree.

The random sampling of points is based on the below equation.

$$\mathbf{x}_{\text{rand}} = \begin{cases} \text{LineTo}(\mathbf{x}_{\text{goal}}) & \text{if } P_r > 1-\alpha \\ \text{Uniform}(\mathcal{X}) & \text{if } \begin{cases} P_r \leq \dfrac{1-\alpha}{\beta} \text{ or} \\ \nexists \text{path}(\mathbf{x}_0, \mathbf{x}_{\text{goal}}) \end{cases} \\ \text{Ellipsis}(\mathbf{x}_0, \mathbf{x}_{\text{goal}}) & \text{otherwise} \end{cases}$$

In the third part of the algorithm, adding node to a tree is performed optimally. This part is called in the second part algorithm.

Algorithm 3 Add Node To Tree

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

For the current node, we find a parent node with the minimum cost to reach.VT and ET denote sets of nodes and edges in the tree, respectively.Note that, when a node is added to the tree, 1) it expands the tree; 2) if it is inside the goal region (Fig 2), a path to x_0 is found (See Section 4.2); 3) we have to update the adjacent grid squares used for grid based spatial indexing (See Section 4.1). Also, it should be noted that we are using a tree structure to build the tree and each node has access to its children and its parent. In other words, we use ET only in the explanation

In the fourth and fifth part of the algorithm, the rewiring of nodes is done. The fourth part rewires the nodes from the random nodes queue i.e. Q_r and the fifth part rewires the nodes from the root nodes queue i.e. Q_s . Both of them are called in the second part of the algorithm.

Algorithm 4 Rewire Random Nodes

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

The node rewiring is same as that done in RRt*. A node is rewired when it gets a lower cost-to-reach. However, in RRT* rewiring just done along the path of the new node from the root node (which is done in fourth part of the algo) whereas here we rewire around the already added nodes. Thus, rewiring is done in xnear if by changing its current parent to x_j, c_i for x_{near} reduces where x_j is x_r and x_s in part four and five, respectively.

In the sixth part of the algorithm, we plan the path of the robot from the current node to either 1) the goal (if the tree finds the goal) 2) closest point to the goal (if the tree does not find the goal)

We use cost function (f_c) to get close tox_{goal} as well as to take a short path on the growing tree. This can trap us in local minima. To prevent trapping in local minima and enable path planning to visit other branches, we plan a k-step path using f_c at each iteration and block the already seen nodes by setting their his to infinity. When path planning reaches a

Algorithm 6 Plan a Path for k Steps

Input: $\mathcal{T}$, $\mathbf{x}_{goal}$
if Tree has reached $\mathbf{x}_{goal}$ **then**
  Update `path` from $\mathbf{x}_{goal}$ to $\mathbf{x}_0$ if the `path` is rewired
  $(\mathbf{x}_0, ..., \mathbf{x}_k) \leftarrow (\mathbf{x}_0, ..., \mathbf{x}_{goal})$
else
  for $\mathbf{x}_i \in (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_k)$ **do**
    $\mathbf{x}_i$=child of $\mathbf{x}_{i-1}$ with minimum f_c=`cost`$(\mathbf{x}_c)$+`H`$(\mathbf{x}_c)$
    if $\mathbf{x}_i$ is leaf or its children are blocked **then**
      $(\mathbf{x}_0', ..., \mathbf{x}_k') \leftarrow (\mathbf{x}_0, ..., \mathbf{x}_i)$
      Block $\mathbf{x}_i$ and Break;
  Update best `path` with $(\mathbf{x}_0', ..., \mathbf{x}_k')$ if necessary
  $(\mathbf{x}_0, ..., \mathbf{x}_k) \leftarrow$ choose to stay in $\mathbf{x}_0$ or follow best `path`
return $(\mathbf{x}_0, ..., \mathbf{x}_k)$

node that could not go any further , we return the planned path and block that node. Then, we update the best already found path if the planned path leads us to a location closer to x_{goal}. However, the agent follows the best path if it leads to some place closer than the current place of the agent.

B. Simulation environment

Here is a brief description of packages we used for the simulation :

1) Python 3.7
2) Numpy
3) ROS 1 (Noetic)

Additionally we tried working with a Unity based simulation provided by the original author. The link to which can be found here. Original Paper Unity implementation

III. Simulation Results

We simulated the project on the gazebo platform with the help of ROS. Prior to that, to get a proper visualization of the map, path and the dynamic goal, we plotted them on a plot using the matplotlib library of python.

In the python plot, after iteration of planning, the tree was plotted in the map. The map consists of the obstacles (along with the buffers on them) and the dynamic goal in yel low, and the robot in blue marked with X. After every plan is executed, we can see the robot moving towards the goal. As the goal changes, the tree gets modified and in every iteration of planning and so does the path of the robot. As per the function of the algorithm, the robot will either reach the goal point or stop after its time exceeds.

For the ROS implementation, we created the same map in a Gazebo world. We have used the turtlebot3 (burger) as our robot. We use the python script implemented above in our publisher-subscriber node. In this node, first the algorithm runs and derives the path which the robot will follow. The turtlebot follows the same path in the Gazebo simulation.

Below is the link of our simulation

Simulation Video Link

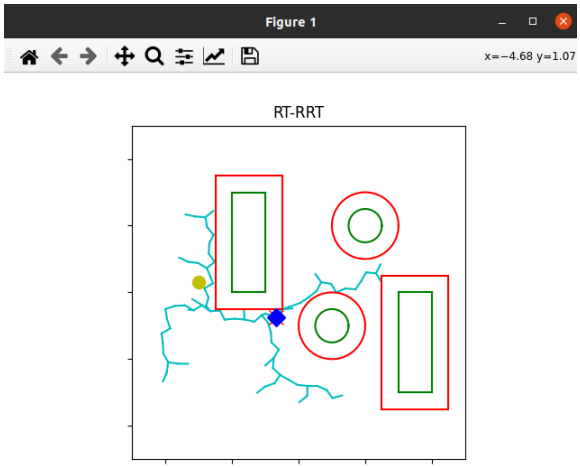

Fig. 1. Live simulation in matplotlib

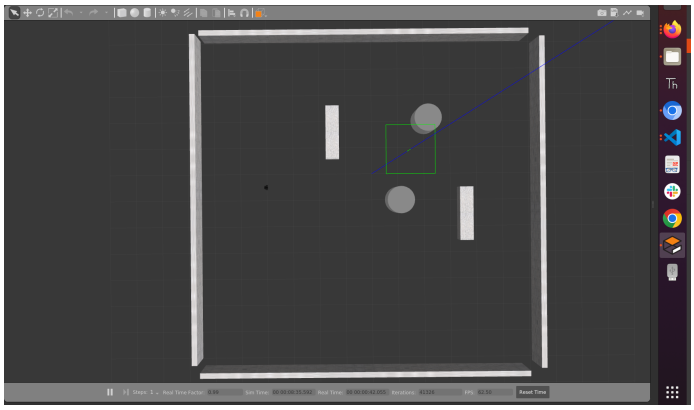

Fig. 2. Gazebo Simulation - top view

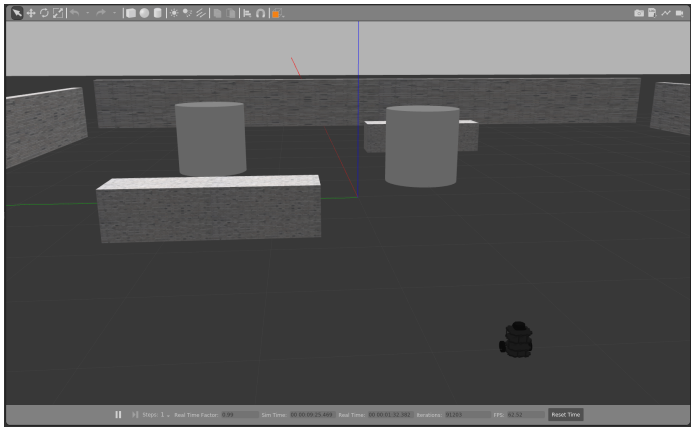

Fig. 3. Gazebo Simulation - lateral view

IV. Conclusion

We have successfully simulated a dynamic motion planning algorithm in a 3D environment. In our opinion the algorithm is very helpful in practical scenarios as it can handle dynamic goal tracking. Although dynamic obstacles were not imple-

mented, the algorithm's ability to handle randomness of the path was sufficient enough to say that the it would work in an environment with dynamic obstacles. Testing this out can certainly be in the future scope of the project.

As RRT* was used in the planning part, it also tries to give the optimal path. Hence it makes motion efficient as well.

A major shortcoming of a real time method is that it cannot guarantee a solution for every case. Even this aspect was observed as there were conditions where there was no solution given by the algorithm.

REFERENCES

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

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

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

[4] BRUCE, J., AND VELOSO, M. 2002. Real-time randomized path planning for robot navigation. In IROS, IEEE.

[5] LUDERS, B. D., KARAMAN, S., FRAZZOLI, E., AND HOW, J. P. 2010. Bounds on tracking error using closed-loop rapidlyexploring random trees. In American Control Conference, IEEE.

[6] STURTEVANT, N. R., BULITKO, V., AND BJORNSSON ¨ , Y. 2010. On learning in agent-centered search. In AAMAS

[7] 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.

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