Path planning with Moving Obstacles

CS 5335 Project Report

Akash Sasikumar, Gail Renee Pinto

Problem Statement

Path planning aims to find a sequence of discrete robot configurations state complying with constraints imposed by the environment and internal dynamics of the system [1]. It is essential for a robot configuration to be able to plan a path to the goal location from the start or current location. While this problem is trivial for a static environment, in real-life applications we see that the operational environment often has many moving obstacles. These obstacles are often moving in unpredictable directions, which makes planning tasks to avoid them difficult. When the obstacle is unknown, the robot will need to be able to dynamically determine a course of action in order to avoid a collision [2].

Sampling based path planning algorithms have become powerful in motion planning framework for mobile robots as well as manipulators to solve complex robotics path planning problem. It is a class of path planning algorithms which are probabilistic in nature. However most of the standard planning algorithms available assume that the environment is static.
For our final project we have focused on the class of RRT based algorithms for both static and dynamic environments.

Algorithm

RRT based algorithm begins with an initial robot configuration as the root node and incrementally samples and grows a tree until it reaches a goal configuration. We have implemented three variants of RRT, i.e. RRT*[3], RRT* for dynamic obstacles, Dynamic-RRT[4]

RRT aims to find a solution as soon as possible. There is a high emphasis on exploration over exploitation. RRT* which is derived from RRT focuses more on optimization.RRT* performs optimization through the process of rewiring nodes which rewires parents and child based on a global optimal cost.

Algorithm 1: RRT*

```
V ← {x_init}, E ← ∅;
for i =1,2,....n do
    x_rand = ← generateSampleNode;
    x_nearest = ← nearestNeighbor(x_rand) ;
    x_new = ← steer(x_rand,x_nearest);
    if CollisionFree(x_nearest, x_new) then
        x_nearindex ← findNearNeighbor(x_new);
        if x_nearindex then
            chooseParent(x_new,x_nearindex);
            rewire(x_new,x_nearindex);
        end
    end
end
```

Algorithm 2: Rewire

```
Parameters: x_new,x_nearindex
for i =1,2,...x_nearindex do
    x_neighbor ← V[i] ;
    if costFromRoot(x_neighbor) <
     (costFromRoot(x_neighbor) +
     dist(x_neighbor,x_new)) then
        x_neighbor ← parent = x_new ;
    end
end
```

This is a standard implementation of RRT* which works well in the case of static obstacles. We have modified this algorithm to work for dynamic obstacles as well.

We use the above mentioned RRT* algorithm for creating a path from the start configuration to the goal configuration with a state of obstacles at the beginning. Once a path has been computed using RRT*, the robot traverses across the path and checks for obstacles. If an obstacle is encountered we clear all the nodes and path that we computed and re-plan again from the current position as the new start position.

Algorithm 3: Modified RRT*

```
Path ← RRT* ;
while Path is not empty do
    if CollisionFreePath then
        nextPoint ← path[0] ;
        moveRobot(nextPoint) ;
        moveObstacle() ;
        path ← path - nextPoint ;
    end
    else
        replanning() ;
    end
end
```

Algorithm 4: Replanning

```
currentPos ← getRobotPosition ;
V ← ∅ ;
Path ← ∅ ;
startPosition ← currentPos ;
RRT*() ;
```

This algorithm is simple to understand however it is computationally very expensive and infeasible especially in a complex environment with constantly moving obstacles. We have implemented another variant of RRT, known as Dynamic RRT which repairs the RRT when there is a collision observed along the path. Instead of abandoning the current RRT entirely, this approach mimics deterministic re-planning algorithms by efficiently removing just the newly invalid parts and maintaining the rest.It then regrows the tree until a new solution is found. [4] Since the start position is constantly changing and we cannot regrow a tree with a moving root, we have set the goal position as the root of the tree and grow the tree backwards. The pseudo code for Dynamic RRT is given below

Algorithm 5: Dynamic RRT

```
q_start ← q_goal;
q_goal ← q_robot;
RRT();
while q_goal ≠ q_start do
    q_goall = Parent(q_goal);
    Move to q_goal and check for obstacles ;
    if new obstacles observed on Path then
        Replanning();
```

Algorithm 6: Replanning

```
invalidateNode();
trimRRT();
RRT();
```

Algorithm 7: trimRRT

```
for i i=1,2,... V.size do
    q_i ← V[i];
    q_p ← q_i→ parent;
    if q_p is invalid then
        q_i ← INVALID;
V_new ← ∅;
for i =1,2,...V.size do
    if q_i is valid then
        V_new ← V ∪ q_i;
V_new ← V;
Create edges between parents and child;
```

Algorithm 8: invalidateNode

```
for each edge e in E do
    if Collision(e) then
        e→child = INVALID
```

Simulation

To demonstrate the implementation of the path planning algorithms, we have used PyBullet library which is built over the Bullet physics engine for simulating our agent in a predefined environment.The environment will have static obstacles and a few moving agents/enemies, which our controlled agent will have to avoid while planning the path to reach the goal.

The environment is a 3-D plane with horizontal and vertical dimensions of 30. The environment contains a set of static and dynamic obstacles. The plane is surrounded by walls on all the sides and the environment contains rigid blocks which act as static obstacles. For our controlled agent we have used predefined husky bot URDF from ROS which we will reference as Husky-Robot.The start position for our controlled agent is (-12,12,0) and the goal is to reach (12,12,0). We have planned a path in the (x, y, θ) configuration space. To move the robot towards the target we have used setJointMotorControl API for setting the velocities of the four wheel joints and calculated the orientation of the robot to face the target.
For the environment with dynamic obstacles, we have used two husky robots with predefined paths. The first robot(Husky-Obs-1) is placed at (0,0,0) and has a vertical motion across the plane. The second robot(Husky-Obs-2) is placed at (12,0,0) and has a horizontal motion across the plane. We have used setJointMotorControl API for setting the velocities of the four wheel joints of each obstacle.

For collision checking we use rayTest API which shoots a ray from point A to point B and returns the obstacle information if it detects an obstacle.
We calculate the wheel position from the base of the robot and convert it to find the position of the wheel in Cartesian coordinates. We calculate the wheel position co-ordinates for both the start position and the next goal position and use rayTest to shoot two rays from the front left and front right wheel.

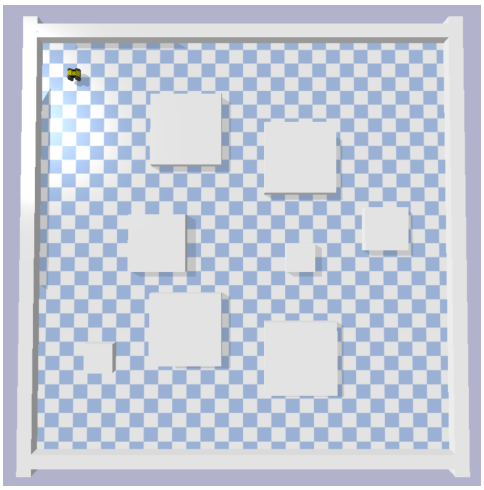

Figure 1: Static Environment

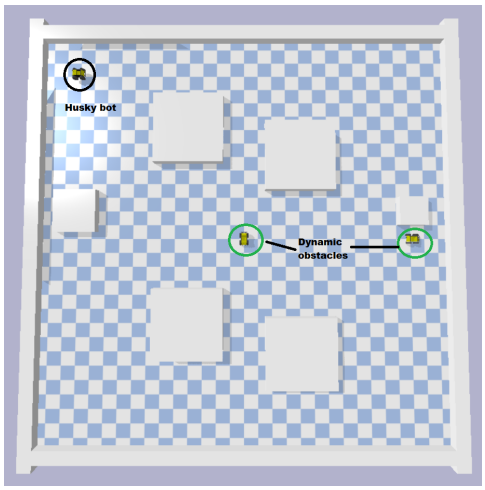

Figure 2: Dynamic Environment

Experiments and Results

In our implementation of path planning algorithms we setup a static and dynamic predefined environments. For our experiments we had defined the step length i.e. maximum distance travelled by the controlled agent in each step as 1. We had defined our search radius as 5.

For our first experiment we did a comparison between RRT and RRT* on our static environment. Since RRT* is an optimal version of RRT, we were expecting RRT* path to be of lower cost which matched our results. We compared results with three experiments of RRT* and RRT with 1000, 2500 and 5000 iterations.
RRT :

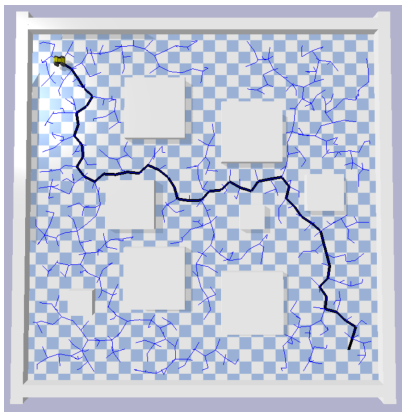

Figure 3: Iteration: 1000

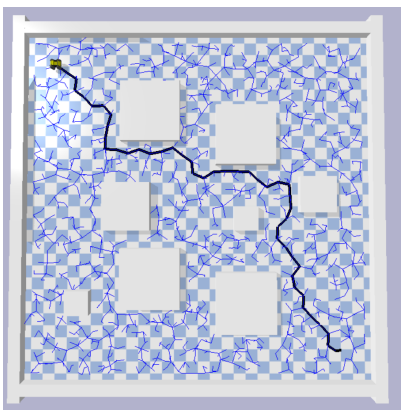

Figure 4: Iteration: 2500

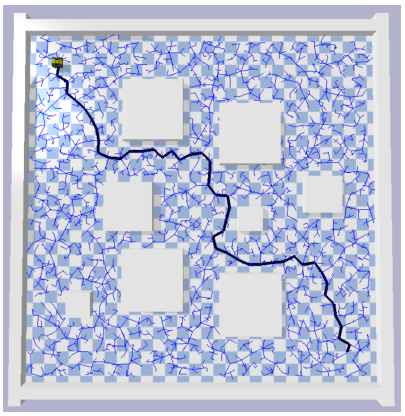

Figure 5: Iteration: 5000

RRT* :

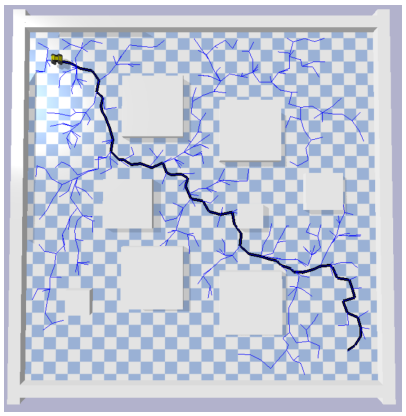

Figure 6: Iteration: 1000

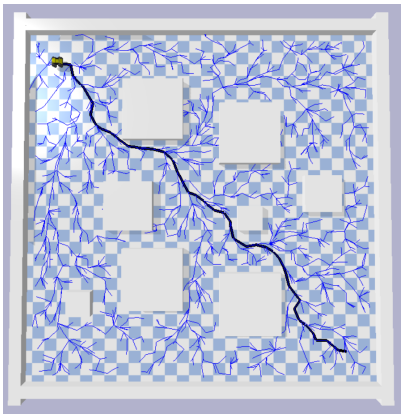

Figure 7: Iteration: 2500

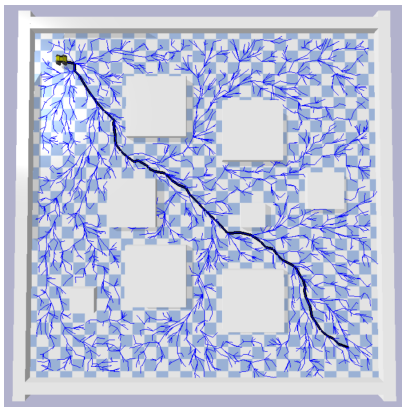

Figure 8: Iteration: 5000

Algorithm	Iteration	Cost
RRT	1000	49.0
	2500	44.77
	5000	44.03
RRT*	1000	44.32
	2500	37.53
	5000	36.02

The straight line cost between the start and the goal position, which would be the most optimal path is around 34 and based on the results of the experiment we can see that RRT* performs better as compared to RRT.

In the case of moving obstacles we have implemented two algorithms : Modified RRT* and Dynamic RRT. Modified RRT* works on discarding the tree and regrowing it when an obstacle is found whereas Dynamic RRT repairs the existing tree.

Similar to the static obstacles we performed three experiments of both Modified RRT* and Dynamic RRT and compared the run time of the two algorithms for maximum iterations set at 1000, 2500 and 5000.

These are the working flows for Modified RRT* with max iterations set at 1000

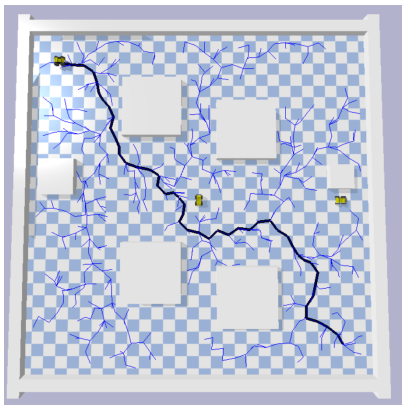

Figure 9: Original Path

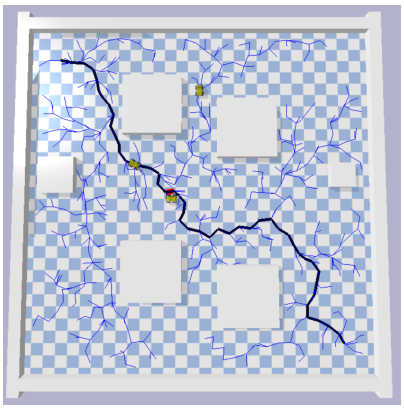

Figure 10: At collision

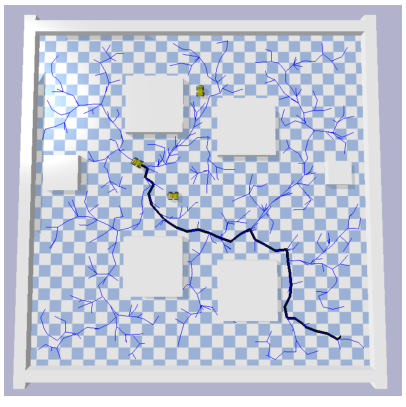

Figure 11: New Path

These are the working flows for Dynamic RRT with max iterations set at 1000

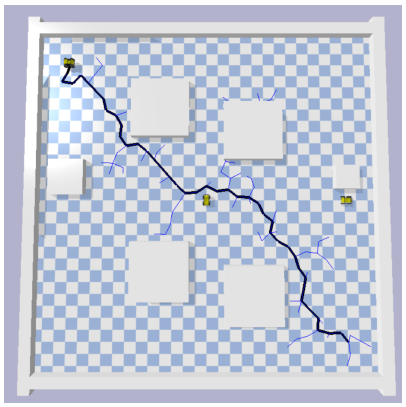

Figure 12: Original Path

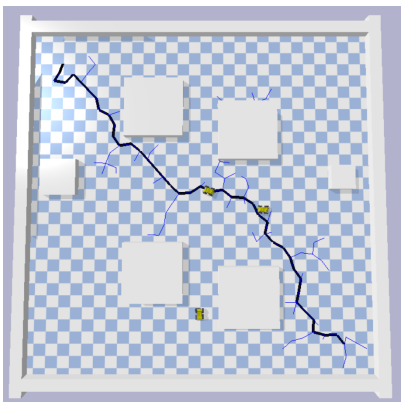

Figure 13: At collision

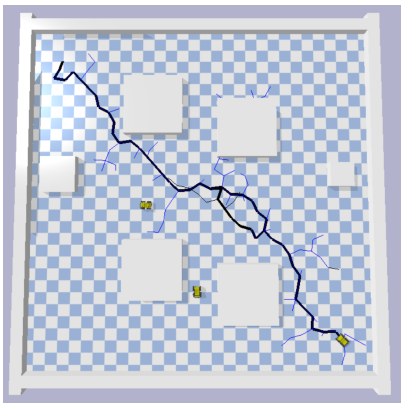

Figure 14: New Path

Algorithm	Iteration	Run time
Modified RRT*	1000	39.59
	2500	115.315
	5000	90.318
Dynamic RRT	1000	22.27
	2500	19.37
	5000	56.26

Based on the observations above and the path taken by both algorithms we can say that Modified RRT* is more computationally expensive and takes a longer time to reach the goal configuration. As the complexity of the environment increases, Dynamic RRT performs tends to find a path faster.

Challenges Faced

In the collision detection procedure used for our experiments we have used rayTest API provided by Pybullet for collision detection. However, we observed that rayTest API showed inconsistent results especially when there were two points just near the border of another obstacle or if two points were entirely inside an obstacle.

Currently the robot detects a collision only if an obstacle trajectory intersects the planned path of the controlled agent. If the obstacle directly collides into our controlled agent, it won't be able to detect that. We tried to avoid this by using other collision detection API provided by Pybullet like getClosestPoints and getContactPoints along with our original rayTest but this didn't give us the required results since these API require knowledge of the other object/obstacle.In our example it was computationally expensive to calculate Closest Points for each obstacle at every time step and thus we did not use these APIs.

Future Work

We could test these algorithms on more complex and dynamic environments which would help us get a better understanding of the efficiency and feasibility of these algorithms.

In our implementation we had used Dynamic RRT for regrowing the tree however since this is a variant of RRT and not RRT*, it does not always result in an optimal path. There are other variants of RRT* such as RT-RRT* and CL-RRT* which also re-plan by regrowing the tree but also provide an optimal path. These algorithms can be used to get a much more efficient path with dynamic obstacles.

References

[1] O. Adiyatov and H. A. Varol. A novel rrt*-based algorithm for motion planning in dynamic environments. In *2017 IEEE International Conference on Mechatronics and Automation (ICMA)*, pages 1416–1421, 2017.

[2] D. Connell and H. M. La. Dynamic path planning and replanning for mobile robots using rrt. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1429–1434, 2017.

[3] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning, 2011.

[4] D. Ferguson, N. Kalra, and A. Stentz. Replanning with rrts. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 1243–1248, 2006.