Sampling Based Motion Planning in a Dynamic Environment

Anshuman Singh
Systems Engineering
University of Maryland
College Park, USA
anshusr3@umd.edu

Haixiang Fang
Mechanical Engineering
University of Maryland
College Park, USA
hfang@terpmail.umd.edu

Jaad Lepak
Aerospace Engineering
University of Maryland
College Park, USA
JLepak@umd.edu

***Abstract*—In autonomous robotics path planning in a dynamic environment is one of the most complex problems that is required to be solved. Popular heuristic based searched algorithms which finds shortest do not perform optimally in real time.Recent efforts have been aimed at solving this problem by suing sampling based algorithms and have had a fair amount of success. This paper build on those efforts by proposing a variation of a popular random sampling based algorithm, RRT*. The proposed algorithm can be implemented to vehicles, robot manipulators for trajectory generation and other agents acting in dynamic environment. A comparison between different sampling based algorithms like RRT, RRT*, Bidirectional RRT,Bidirectional RRT* has also been studied and presented in this paper.**

***Index Terms*—RRT*, Sampling based path planning, dynamic environment**

I. Introduction

In robotics, path planning and trajectory generation has always been one the most researched area in the last decade. Autonomous robotics are expected to reliably perform path planning among obstacles and humans as one of their fundamental tasks i.e. to navigate the robot in an environment while selecting an optimal and collision free path. Existing heuristic, graph-based search algorithms like A* and Dijkstra, work well for static environment however they do not provide us with optimal solution in real time and often become complex in nature for higher dimensional problems. Due to these limitations we use sampling based algorithms for path planning in dynamic environment which often is the case while dealing with real world problems. RRT and RRT* are one of the most popular sampling based planning algorithms and have been used extensively in the past few years. In this paper we will introduce a variation of RRT* where re-planning occurs only in an area blocked by the obstacle. The Rapidly Exploring Random Tree star (RRT*) method will be used to find the optimal path for the robot traveling from the start to the target and avoid static and dynamic obstacles in the environment. The overall strategy is to calculate the robot's optimal path using sensors with a limited range to detect static and dynamic obstacles blocking its path. When an obstacle is detected, the route is re-calculated however, the re-planning takes part only in a small region. The robot steers off the optimal path, avoids collision by moving around the obstacle, and then continues on the optimal path toward the target.

II. Background

Planning algorithms can be roughly classified into 2 major categories: Search based planning algorithms and Sampling based planning algorithms.The basic idea of Search-based planning algorithms is to discretize the configuration space i.e. to represent the configuration space as a grid of regularly sized grid cells. The initial position and final position of robot is known and search is run on the grid to find the path from the initial position to the final position. Some of the most used Search-based planning algorithms are Dijkstra and A* (A star) algorithm.

Dijkstra uses a cost to come function. When searching through the map, each point along a path calculates the cost to come from the starting point to the current point. As Dijkstra generates different paths, there can be redundancies where multiple paths reach the same point. In order to decide the optimal path, Dijkstra chooses the one with the least cost to come.

A* uses the sum of a cost to come and cost to go function. Cost to go calculates the least cost from the current point to the goal point. When choosing the optimal path, the heuristic function for A* is designed to choose paths with the lowest sum of cost to come plus cost to go. In most scenarios the environment is barely static. Hence, we shall investigate sampling-based planning methods Sampling based planning algorithms randomly sample different robot configuration and check if they are valid or invalid and finally form a graph consisting of the valid configurations (not colliding with the obstacle) of the robot [1] [2]

In contrast to Search based algorithms, Sampling based algorithms have proven to be useful for high dimension motion planning. It should be noted that most sampling-based algorithms while not complete, are probabilistically complete i.e. if the number of samples (randomly generated vertices) tends to infinity then the probability of finding a path tend to 1 [3] The following steps outline the general steps of a sampling algorithm: Select the starting vertex,Randomly generate a vertex ,Check if the generated vertex is inside of an obstacle. If it is then a new the previously generated vertex is discarded and new vertex is randomly generated again When a valid vertex is found, an edge is added between the valid

vertex and it's neighbors This process is related till we get a graph which has a path between the start and goal vertex. Now that we have a graph, a search-based algorithm like A* is run to find the optimal path on the graph.

Algorithm 1: General overview of Sampling based algorithm

Input: q_{init}
Output: $Tree(Edge, Vertex)$
$Tree = Init(X, start)$
while $ElapsedTime() < t\ NoGoalFound(G)$ **do**
  newpoint = StateToExpandFrom(Tree)
  newsegment = CreatePathToTree(newpoint)
  if $ChooseToAdd(newsegment)$ **then**
    Tree=Insert(Tree,newsegment)
  end
end

where q_{init} is the imitial or start node X is a d dimensional state space. G $\subset$ X state space considered as goal and t is the time allowed

A. Literature Review

RRT is a widely used sample-based algorithm to solve problems with holonomic and kinodynamic constraints. The general algorithm followed by RRT is described above however one should note the new generated vertex is joined to the closest existing vertex. RRTs are able to solve higher dimensionality problems easily however the path mostly is not optimal. In fact, it can be proved that the best path made by RRT is almost always is the non-optimal one. [4] An extensive amount of research has been performed to increases the efficiency of the RRT algorithm. One such method is the bi-directional RRT where RRT is performed from the start node and the goal node simultaneously which increases the speed at which single query solution is found. Another extremely famous method is RRT* which provides a more optimal solution than RRT. RRT* has gained popularity in the recent years as among the robotics community due to its simplicity and probabilistic complete nature i.e. it provides a sub-optimal solution without having complete information about the obstacle space. RRT* is similar to RRT except, a cost function is associated with every vertex. Whenever, a new random vertex is generated, neighbors of the generated node are selected and the generated node is joined to an existing node (neighbors) with gives the lowest total cost to the newly generated node. Next, the neighbors are again checked to verify if there exists another "less costly" path to any of the neighbor nodes through the newly generated node. This is called rewiring of the map and restructures the map. This restructuring of the map produces a more optimal path than the RRT which ensures the sub-optimality of the algorithm. Even though RRT* does provide a more optimal path, it takes a significantly longer time to execute. [5] [6] Furthermore, a survey paper talks about variations in RRT* to overcome the drawbacks of the traditionally used RRT* algorithm which yields non-optimal cost values. [9] The selection of a variation of RRT* is based on similar concepts such as type of environment information available, the structure of the tree and the constraints managed by approach. One variation is transition based RRT* (T-RRT*) which focuses on continuous configuration-cost space. [10] The approach integrated transition test-based functions used in T-RRT to address the issue of path quality with respect to a given criterion. Another variant of RRT* proposed by Moon et al. [11] presented a kinodynamic variant of RRT* called Dual-Tree RRT (DT-RRT). DT-RRT manages two trees called state tree and work-space tree. At first, workspace tree explores targeted environment without considering any physical constraints. Then, state tree generates trajectories from work-space tree nodes using kinematic and dynamic constraints. Additionally, random obstacles have been introduced in order to replicate the occurrence of dynamic obstacles [12] in the real world. An extension to a leading real-time planner, LSS-LRTA*, is introduced to replace existing heuristic search methods for real-time planning in dynamic environments in case of failing in the high dimensional state space.

III. Proposed Methodology

In our variation of RRT* we start by using bidirectional RRT* to find a path from a start node to a goal node while considering the obstacles are static. This path is called the Initial Path (shown in red line) and stored in a list of tuples. While looking ahead (green dotted circle of radius r), on the initial path, the robot moves towards the goal. If the robot senses an obstacle on the initial path inside the radius, all the nodes of the initial path which are on the boundary of the obstacle and inside the obstacle (shown in with dark blue line) are removed from the initial path. Next, we apply RRT to to the immediate points outside of the obstacle and find a new "sub-path" (shown with a green line). The robot takes the "sub-path" to avoid the obstacles and then tries rejoin the initial path to continue towards the goal. If the robot encounters the obstacle while it traverses the "sub-path" it again discards the nodes inside the obstacle and finds a new sub-path to rejoin the initial path.

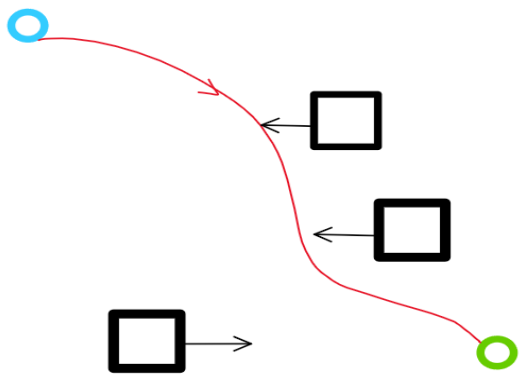

Fig. 1. Inital Path found by Bi-DRRT*

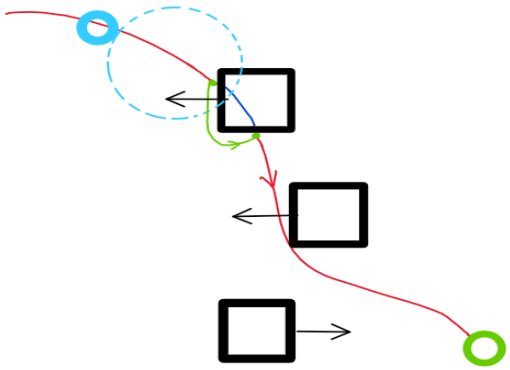

Fig. 2. Invalidating nodes and finding sub-pth

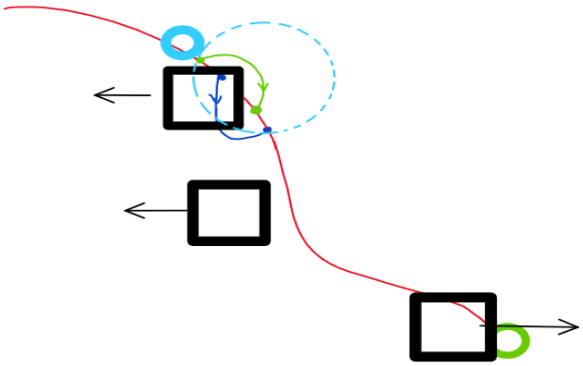

Fig. 3. Invalidating the sub-path due to movemnt of obstacle and finding new subpath

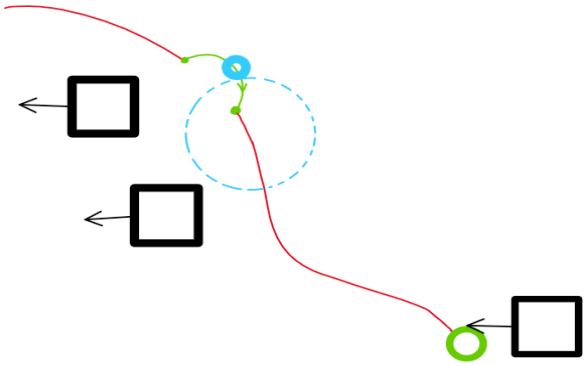

Fig. 4. New Path

A. *RRT**

The Rapidly Exploring Random Tree star (RRT*) method will be used to find the optimal path for the smart torpedo traveling from the submarine to the target and avoid static and dynamic obstacles in the environment. The overall strategy is to calculate the torpedo's optimal path using sensors with a limited range to detect static and dynamic obstacles blocking the torpedo's path. When an obstacle is detected, the route is re-calculated. The torpedo steers off the optimal path, avoids collision by moving around the obstacle, and then continues on the optimal path toward the target. This algorithm is broken down into 6 parts and is detailed below.

Algorithm 2: General overview of RRT*

Input: q_{init}
Output: $Tree(Edge, Vertex)$
1 $Tree \longleftarrow$ Initialize Tree()
2 **for** *i in range (1,N)* **do**
3 $q_{random} \longleftarrow$ GenerateRandomSample(i)
4 $q_{nearest} \longleftarrow$ NearestNeighbour(q_{random}, Q_{near},Tree)
5 $q_{min} \longleftarrow$ ChooseParent($q_{random}, Q_{near}, q_{nearest}, \Delta q$)
6 $Tree \longleftarrow$ InsertNode(q_{min}, q_{random}, Tree)
7 $Tree \longleftarrow$ Rewire(Tree,$Q_{near}, q_{min}, q_{random}$)
8 **end**

where

- q_{random} is the randomly generated node
- Q_{near} is the **list** of all nodes in the neighborhood of the randomly generated node
- $q_{nearest}$ is the nearest node to the randomly generated node
- Δq is the threshold at which nodes should be generated
- q_{min} is the neighbour with the lowest cost

Line 1 : We initialize the algorithm by providing a starting node and inserting it into the tree to initialize the root of the tree.

Line 2: For N number of iterations do the following

- Line 3: we keep generating random nodes
- Line 4: During each iteration, we use the Nearest-Neighbour() function to find the nearest neighbour
- Line 5:In RRT* instead of finding the nearest neighbour like RRT, we find the best neighbour q_{min} which is the lowest cost neighbor.The ChooseParent() function in line 5 returns the best neighbour q_{min}
- Line 6:Next, we insert that node into the tree
- Line 7: Change the tree structure based upon the nodes which are inserted into the tree. This rewiring of the tree results in sub-optimal solution for RRT*

where

- q_{near} is **one of the node** in the neighborhood of the randomly generated node i.e. a subset of Q_{near}
- $CurrentCost_{min}$ is the current best cost
- $PathFinder$ return a path from the nearby node, q_{near} to q_{random}
- q_{path} is the path returned by $PathFinder$ item $Cost_{new}$ is new cost of the path from the root node to the randomly generated node,q_{random} via one of the near nodes,q_{near}

The Algorithm 2, ChooseParent(), selects the best parent from the neighborhood of randomly generated nodes.

Algorithm 3: ChooseParent

Input: $q_{random}, Q_{near}, q_{nearest}, \Delta q$
Output: q_{min}

```
1  q_min ⟵ q_nearest
   CurrentCost_min ⟵ Cost_{q_nearest} + Cost_{q_random}
2  for q_near ∈ Q_near do
3      q_path ⟵ PathFinder(q_near, q_random, Δq)
4      if q_path(ObstacleFree) then
5          Cost_new ⟵ Cost_{q_near} + Cost_{q_path}
6          if Cost_new < CurrentCost_min then
7              CurrentCost_min ⟵ Cost_new
8              q_min ⟵ q_near
9          end
10     end
11 end
12 return q_min
```

Line 1: The nearest neighbour is considered as the minimum cost node

Line 2: The cost of reaching to the randomly generated node through the nearest node is stored in CurrentCost

Line 3: We then search through all nodes in the neighbourhood of the randomly generated node

Line 4: Returns a path from the nearby node, q_{near}(subset of Q_{near}),to randomly generated node

Line 5,6,7: If this path is obstacle free and has a lower lost than the Current minimum cost,$CurrentCost_{min}$,then the nearby node,q_{near}, becomes the best neighbor, q_{min} and this cost becomes the best cost $CurrentCost_{min}$ (lines 7-9 of Algorithm 4).

When all nearby nodes have been examined the function returns the best neighbor. The new random node is inserted into the tree using q min as the parent

As described in Algorithm 3, the Rewire() function called in line 7 of Algorithm 1 will change the tree structure based on newly inserted node q_{rand}.

Algorithm 4: Rewire

Input: $Tree, Q_{near}, q_{min}, q_{random}$
Output: $Tree$

```
1  for q_near ∈ Q_near do
2      q_path ⟵ PathFinder(q_near, q_random)
3      if q_path(ObstacleFree) and
         Cost_{q_random} + Cost_{q_path} < Cost_{q_near} then
4          Tree ⟵ Reconnect(q_random, q_near, Tree)
5      end
6  end
return Tree
```

where

- Reconnect() is a function used to rewire an edge based on the newly updated parent node.

Line 1: Choose a nearby node within the given neighborhood set

Line 2: Returns a path from the nearby node to the randomly generated node

Line 3: If the path is obstacle free and total cost of this path is lower than the current parent of q_{near}

Line 4: The tree will then remove the existing edge to the current parent of q_{near}, and rebuild a new edge to update q_{random} as the parent of q_{near}.

The ChooseParent() and Rewire() functions together make sure the paths generated will be asymptotically sub-optimal since they are always minimizing the costs to reach the nodes within the tree. Comparing to the RRT algorithm which has tree branches moving in all direction, the ChooseParent() function ensures edges created in RRT* always moving away from the satrting point. Additionally, the Rewire() function will avoid unnecessary vertices on any discovered path.

B. Dynamic Re-planning

Considering the complexity of the real world, random obstacles have been introduced in our project in order to replicate the occurrence of the dynamic environment. When the obstacle position is unknown or it is moving in unpredictable directions, the robot will need to be capable of determining the path dynamically in order to avoid a collision.

Algorithm 5: ExecutePath

```
1  SetObsDentination(numObs)
2  SetObsVelocities(numObs)
3  SetRobotDestination()
4  SetRobotVelocity()
5  while RobotLocation! = GOAL do
6      UpdateObsLocatioon(numObs)
7      UpdateRobotLocation()
8      if Replan then
9          DoReplan()
10     end
11 end
```

In order for obstacles to move in the configuration space, a graph is generated to provide the paths between those static obstacles. The vertices are the intersections of the paths.

Line 1,2: The obstacles are initialized by placing at random vertices. It will then choose a random adjacent vertex and begin to move to that vertex with a initial speed

Line 3,4: Set the goal location and initialize a velocity for the robot

Line 5: After the optimal path is generated by the search tree, the robot follows the nodes in the optimal path with required velocity vectors and keeps moving until reaches the goal node.

Line 6: If any obstacle reaches the vertex, a new vertex will be chosen. The obstacle will keep moving in the new direction.

Line 7: Keep updating the location to check if the path is feasible. The UpdateRobotLocation() function will be explained in Algorithm 6.

Line 8,9,10: If the robot encounters a random moving obstacle, a replan event occurs as described in Algorithm 5 below.

Algorithm 6: DoReplan

Output: $Tree$

1 InvalidateNodes()
2 GetReplanGoalLocation()
3 SetReplanSamplingLimits()
4 RRT(q_{robot})
5 Rewire($Tree, Q_{all}, NULL, q_{robot}$)
6 SetReplanPath()

Line 1: This is executed when a moving obstacle comes in the way of the robot, therefore a new route must be planned. The current path nodes are invalidated as an optimal path in the tree

Line 2: New nodes are chosen to navigate the robot around the moving obstacle. Nodes further from the robot than the obstacle are chosen. Nodes closest to the obstacle and connect to the optimal path make up the new route

Line 3: A region area encompassing the robot, moving obstacle, and new nodes is created

Line 4: The tree is expanded within the region area defined in line 3

Line 5: The optimal path is calculated

Line 6: The new optimal route is updated.

When the search tree replanning is complete and the new best path to the goal location is found. ExecutePath() will begin again. If the robot hits another moving obstacle, the replanning will be repeated. However, the replan goal location will always be a node on the original optimal path.

The robot's movements are tracked using Algorithm 6. This is where the robot's sensors anticipate moving obstacles potentially blocking the optimal path. The location of the robot is kept track using it's velocity. Then, nearby obstacles are looked at. The algorithm checks if any static obstacles are blocking the robot's sensors from seeing moving obstacles. If the visible range is clear and a moving obstacle is detected to cross the robot's path, then part of the total optimal path must be re-routed. Algorithm 5 is executed in line 13 of Algorithm 6 and the optimal path re-routes as close to the moving obstacle without collision, then returns the robot to the overall optimal path.

Line 1: Update robot location using robot velocity to get new position

Line 2: If robot reached the destination, then

Line 3: Update new robot destination with the next path to explore

Algorithm 7: UpdateRobotLocation

1 $robotLocation \longleftarrow robotLocation + robotVelocity$
2 **if** $robotLocation$ == $robotDestination$ **then**
3 $\quad robotDestination \longleftarrow GenerateNextPathLocation()$
4 $\quad SetRobotVelocity()$
5 **end**
6 **while** *obstacleIndex* $<$ *numObs* **do**
7 $\quad$ obsDistance $\longleftarrow GetDistance$(robotLocation,...
8 $\quad$ ...$ObsLocation(ObsIndex)$)
9 $\quad$ **if** *ObsDistance<robotRange* **then**
10 $\quad\quad$ obs$_{path}\longleftarrow$PathFinder($robotLocation, obsLocation$)
11 $\quad\quad$ **if** *ObstacleFree(*Obs_{path}*)* **then**
12 $\quad\quad\quad$ **if** *IsPathBlocked(obsIndex)* **then**
13 $\quad\quad\quad\quad Replan \longleftarrow$TRUE
14 $\quad\quad\quad$ **end**
15 $\quad\quad\quad$ SetObsVisible(obsIndex)
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 **end**

Line 4: Set robot velocity value

Line 5: End of IF loop from line 2

Line 6: A number of static and moving obstacles will be looked at, ranging from the starting index number to the total number of obstacles. While the obstacle's index number is less than the total number of obstacles, do:

Line 7: Calculate the distance to the obstacle using the robot's current location ...

Line 8: ... and current obstacle's location

Line 9: This is where we check for possible collisions between the robot and obstacles. The robot has sensors to check for obstacles moving in it's way. However, it first must check if static obstacles will block it's line of sight from detecting moving obstacles. If the distance to the obstacle is within the sensor's range of the robot detection system, then

Line 10: Update the obstacle path variable to the robot's current projected path

Line 11: If no static obstacles are blocking the robot's line of sight to detect moving obstacles, then

Line 12: If the moving obstacle will block the robot's projected path, then

Line 13: Re-plan the robot's route

Line 14: End of IF loop from line 12

Line 15: Find the next obstacle visible from the robot's sensors

Line 16: End of IF loop from line 11

Line 17: End of IF loop from line 9

Line 18: End of WHILE loop from line 6

IV. Implementation

A. Search Algorithms Comparison

We considered several search algorithms that could be used to find an optimal path across the whole map:

1. RRT - Searches randomly populates nodes around the map.

2. RRT* - An optimized version of RRT, helping to provide a shorter path.

3. Bi-RRT* - Performs two RRT* searches, one starting at the robot's initial position and one at the goal. The two searches continue throughout the map until the end points are within a threshold. The two paths are joined to provide an optimal path.

Algorithm 8: Bi-Directional RRT*

Input: q_{init}
Output: q_{goal}
$V_a \longleftarrow Q_{init}$;$V_b \longleftarrow q_{goal}$
$T_a \longleftarrow (V_a,E_a)$;$T_b \longleftarrow (V_b,E_b)$
$\tau_{best} \longleftarrow \infty$
for $i \longleftarrow$ *0 to N* **do**
  $q_{rand} \longleftarrow$ RandSample(i)
  $q_{nearest} \longleftarrow$ NearestVertex(q_{rand},T_a)
  $q_{new} \longleftarrow$ Extend($q_{nearest}$,q_{rand})
  $Q_{near} \longleftarrow$ NeighbouringVertices(q_{new},T_a)
  L$\longleftarrow$ ListSorting(q_{new} ,q_{parent},T_a)
  if q_{parent} **then**
    T_a (V_a,E_a) $\longleftarrow$ VertexInsert(q_{new},q_{parent},T_a)
    $E_a \longleftarrow$ RewiringVertices(q_{new},L,E_a)
  end
  $z_{conn} \longleftarrow$ NearestVertex(q_{new},T_b)
  $\tau_{new} \longleftarrow$ Connect(q_{new},q_{conn},T_b)
  if $\tau \neq \Phi$ *and Cost*(τ_{new}) $<$ *Cost*(τ_{best}) **then**
    $\tau_{best} \longleftarrow \tau_{new}$
  end
  SwapTrees(T_a,T_b)
end
return T_a,T_b =(V,E)

Algorithm 9: Connect

Input: q_{init}
Output: q_{goal}
$q_{new} \longleftarrow$ Extend(q_1,q_2)
$q_{near} \longleftarrow$ NeighbouringVertices(q_{new},T_b)
List $\longleftarrow$ ListSorting(q_1,q_{near})
$q_{parent} \longleftarrow$ PickBestParent(List)
if q_{parent} **then**
  E$\longleftarrow$(q_{parent},q_1)
  $\tau_{tree} \longleftarrow MakePath$(q$_{parent}$,$q_1$)
  return τ_{tree}
end
return NULL

where L is a list of the total cost of q_near in ascending order. In Bi-Direction RRT* we initialize 2 trees T_a and T_b. T_a is initialized by q_init as the root vertex and T_b is initialized with q_goal as the root root vertex. Bi-directional RRT* works exactly like RRT* till the rewiring where a node is inserted to T_a. Next, we search for a for a nearest vertex (q_conn) from Tree b to to node q_new . Next the connect function returns a path τ_new between T_a and T_b having a cost $Cost(\tau_{new})$. If $Cost(\tau_{new})$ is less than the previous best cost $Cost(\tau_{best})$ then $Cost(\tau_{best})$ is replaced by τ_new and then T_a and T_b are swapped.

First, we looked at the entire map with static obstacles. Three algorithms were used respectively to find a path from start to goal across the whole map, and the solution was labeled as the original optimal path. Then the robot begins to move. A square region around the robot, about the same size as an obstacle, is used to test detection when a robot is on route to collision. When an obstacle collides with this region, the robot simply runs the same algorithm corresponding to the original again. The start is the robot's current position and the goal is the original goal at the end of the map.

B. Re-planning Optimization

Next, we focused on re-planning only a portion of the optimal path, as opposed to the whole path. During movement, the region in front of the robot is used to detect obstacles. We adjusted the detection region to be a circle in front of the robot. When the region intersects an obstacle, a second RRT search is performed. This time, the RRT searches for a diverging path from the robot's current position to a temporary goal. This temporary goal is located 10 nodes along the original optimal path. The robot then travels on this diverging path to avoid the obstacle and continue on the original optimal path. The robot only decides one divergent path at a time.

V. Result

A. Comparative Study of Various Search Algorithms

We performed a comparison between RRT, RRT* and Bi-directional RRT* based on the basic re-planning method which is using a square region to detect and re-plannig to the final goal. The path exploring for the three algorithms as well as the re-planning method are shown respectively in the figures below.

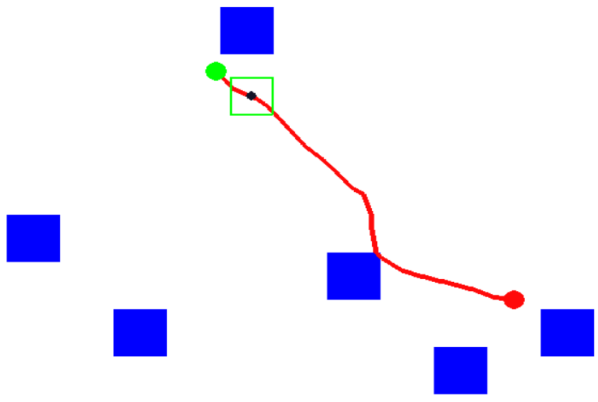

Fig. 5. Basic Re-plan Method

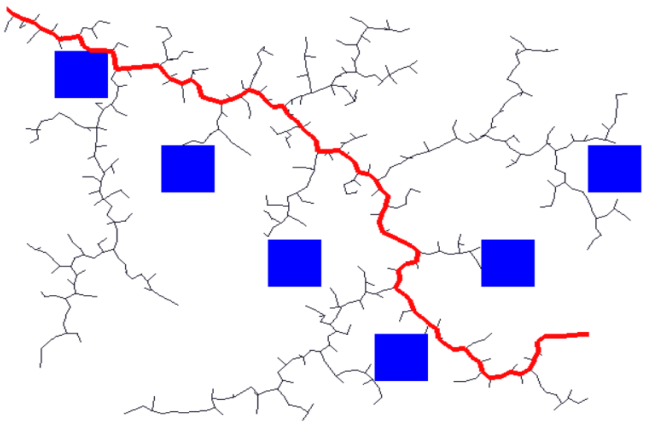

Fig. 6. RRT

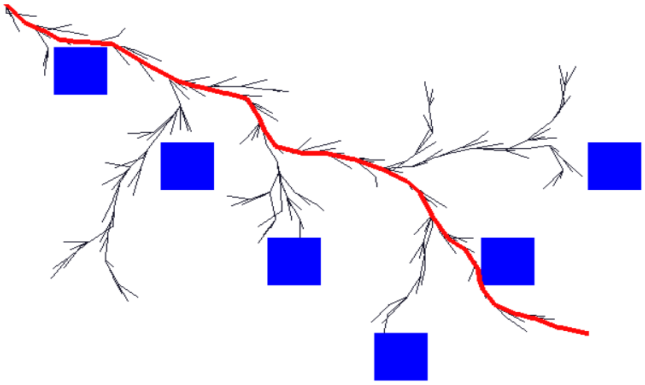

Fig. 7. RRT*

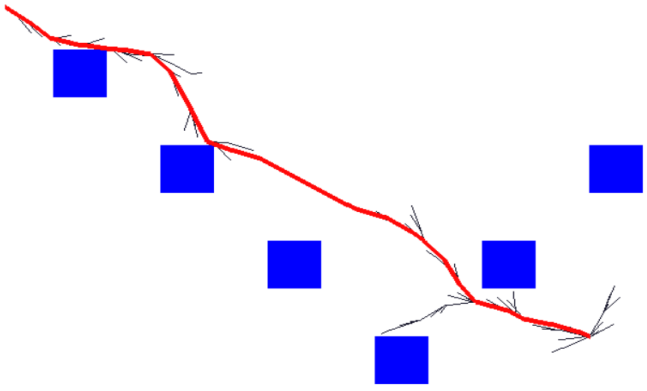

Fig. 8. BI-Directional RRT*

We evaluated the performance of three algorithms from 4 aspects, which are time taken to search the path, total time cost for robot movement, final path length and the total number of visited nodes in the map. Considering the random nature of the sampling-based method, we ran each algorithm 10 times and took the average to get the data as shown in the table below.

TABLE I
PERFORMANCE OF RRT,RRT*, BIDIRECTIONAL RRT*

Algorithm	RRT	RRT*	Bi-RRT*
Time Taken to Find the Path (s)	0.31	1.12	0.17
Total Time Cost(s)	59.07	60.68	19.73
Path Length (units)	2292.27	1977.52	1894.25
Nodes Explored	2016.10	1710.40	333.30

From the comparison, we found out that RRT finds the path in a reasonable amount of time but it is neither smooth nor optimal. RRT* finds a better path regarding the path length and visited nodes than RRT but takes more time to execute. Sometimes the program even ran out of 3000 nodes to get the result. On the contrary Bi-directional RRT* offered the best trade-off between time taken and path cost. The path found was relatively optimal and it took significantly less time and less exploration than the other two algorithms.

B. Bi-RRT and RRT Algorithms Combination and Re-plan Optimization*

Taking advantages of the three algorithms above, we came up with a new algorithms combination, which first using Bi-RRT* to quickly find the optimal path through the whole map and then utilizing RRT to re-plan if the robot hitting the obstacle along the path. Furthermore, we implemented the optimized re-plan method as setting a circle detection region ahead of the robot and re-planning to a temporary goal instead of the final goal. The mechanism is shown as Fig 9 below. The original optimal path is shown in red. Search region is a green hollow circle and divergent paths are green.

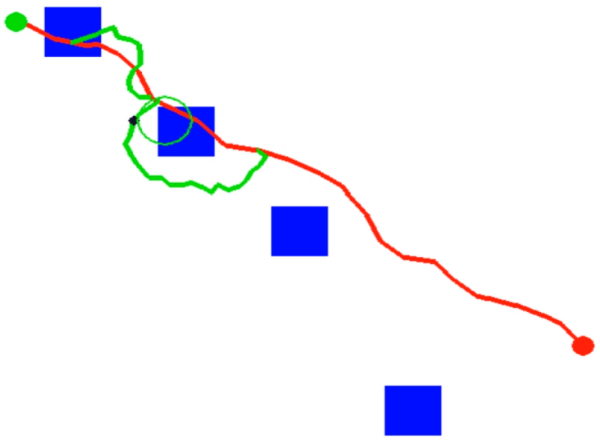

Fig. 9. Algorithms Combination and Advanced Re-plan Method

From the data showing in the table below, our new path planning algorithm reached the least cost in all aspects and achieved the optimum.

TABLE II
PERFORMANCE OF RRT AND BI-RRT* COMBINATION ALGORITHM

Total Time Cost(s)	Path Length (units)	Nodes Explored
5.27	230.42	501.10

VI. CONCLUSIONS AND FUTURE WORKS

We accomplished what we set out to in the initial proposal, but there are improvements that can be made. Fig 10 shows the robot diverging on a path that eventually collides with the obstacle. This can be avoided if the robot was able to anticipate the obstacle's direction. If we know the obstacle's direction, we can continue to run an RRT search until it gives us an optimal path that is behind the obstacle. Thus, the obstacle will continue to move away from the robot as the robot travels along the diverging path, avoiding collision. In our future research, we will address the above issue and implement the algorithm on TurtleBot and simulate it in ROS, Gazebo. Moreover, we also plan to use Machine Learning to help the robot navigate faster in unseen environments.

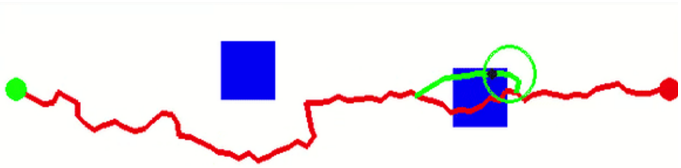

Fig. 10. Re-planned path passing through obstacle

VII. PROBLEM ENCOUNTERED

As we proposed, the re-plan mechanism should be triggered when the obstacles touches the detection region as well as blocks the original path. Then we got a problem with determining the size of the search region. If the region is too small, re-plan couldn't find a path because the obstacle blocks too much area. If the region is large enough, we still had to consider two scenarios regarding triggering the re-plan. Based on the obstacle location shown in the Fig 11 below. If the re-plan is triggered when the detection region interacts with the top obstacle, it would be too late to avoid it; if re-plan when the bottom obstacle affects the detection region, it would be too early or unnecessary since the obstacle might already have passed by when the robot actually gets there. Therefore, in order to simplify the problem, we decided to define a relatively small search region and let it be triggered as long as the obstacle touches the boundary even if not touching the red original path.

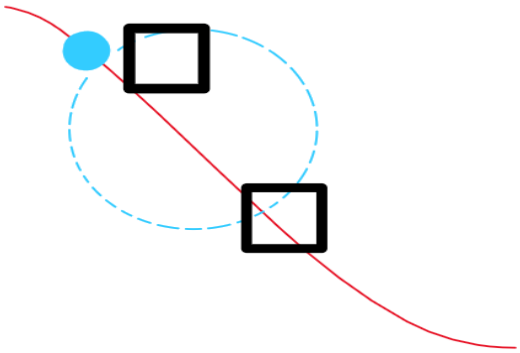

Fig. 11. Re-plan trigger depends on obstacle locations

ACKNOWLEDGMENT

We would like to thank Dr. Reza Monferadi for his constant guidance, motivation and his support. We are grateful for his feedback and his discussions about implementation of path planning in real life.

REFERENCES

[1] Choset H, Lynch KM, Hutchinson S, Kantor G, Burgard W, et al. 2005.Principles of Robot Motion: Theory, Algorithms, and Implementations.Cambridge, MA: MIT Press

[2] LaValle SM. 2006. Planning Algorithms. Cambridge, UK: Cambridge Univ. Press

[3] Ladd AM, Kavraki LE. 2004. Measure theoretic analysis of probabilisticpath planning. IEEE Trans. Robot. Autom. 20:229–424

[4] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning. Robotics: Science and Systems (RSS), 2010

[5] S. Karaman and E. Frazzoli, "Optimal kinodynamic motion planning using incremental sampling-based methods," in 49th IEEE Conference on Decision and Control (CDC), Dec 2010, pp. 7681–7687

[6] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller, "Anytime motion planning using the rrt*," in 2011 IEEE International Conference on Robotics and Automation, May 2011, pp. 1478–1483

[7] Pearl, Judea (1984). Heuristics: intelligent search strategies for computer problem solving. United States: Addison-Wesley Pub. Co.,Inc., Reading, MA. p. 3

[8] Kuffner, James J., and Steven M. LaValle, "RRT-connect: An efficient approach to single-query path planning," Robotics and Automation,2000. Proceedings. ICRA'00. IEEE International Conference on. Vol. 2.IEEE, 2000.

[9] Jarad Cannon and Kevin Rose and Wheeler Ruml, "Real-Time Motion Planning with Dynamic Obstacles," Proceedings of the Fifth Annual Symposium on Combinatorial Search

[10] Devaurs, T. Simeon, and J. Cortes, "Optimal path planning in complex cost spaces with sampling-based algorithms", IEEE T AUTOMSCI ENG, vol. 13, pp. 415-424, Apr 2016.

[11] Moon, and W. Chung, "Kinodynamic planner dual-tree RRT (dt-RRT) for two wheeled mobile robots using the rapidly exploring random tree", IEEE T IND ELECTRON, vol. 62, February 2015

[12] I. Noreen, A. Khan, Z. Habib, "Optimal Path Planning using RRT*based Approaches: A Survey and Future Directions," IJACSA, Vol. 7, No.11, 2016