Field-aided RRT*

Ishan Chadha
College of Computing
Georgia Institute of Technology
Atlanta, USA
ichadha3@gatech.edu

Mudit Gupta
College of Computing
Georgia Institute of Technology
Atlanta, USA
mgupta303@gatech.edu

I. Introduction

RRT* works well for static, fully observable environments, but when the environment is partially observable or changing, then replanning must be achieved efficiently and accurately. Our goal is to investigate improvements to RRT for partially-observable environments by incorporating ideas from other path-planning domains.

II. Background and Related Works

A. RRT and Variants

A rapidly-exploring random tree (RRT) is an algorithm that randomly samples a certain number of nodes in a nonconvex search space and then builds a tree from the start node to the goal node along these samples. In order to connect the tree at each time step, the tree is extended towards either the nearest node that does not cause a cycle in the tree or a point that lies on the path between the current node and the nearest node that is limited by some growth factor.

RRT* expands on the concept of RRT in two ways: it creates a ball of volume $V = \gamma log(n)/n$ where γ is a constant and n is the number of samples taken, and it performs a smoothing procedure to connect nodes that are not adjacent but can be connected without the tree intersecting an obstacle. These improvements allow RRT* to achieve asymptotic optimality while simultaneously creating a path that can be more easily traversed due to the removal of zigzagging edges between proximal nodes in the tree.

B. A* Graph Search and Variants

A* is an informed search algorithm that incorporates the cost of traversing a node n, g(n), with the cost left to get to the goal, h(n). The cost to get to the goal, h(n), is approximated via an admissible heuristic. Seen nodes are traversed based on a priority queue ordered by $f(n) = g(n) + h(n)$, and a path from the start to the goal is constructed.

First described by Koenig and Likhachev in the paper Incremental A* [1], Lifelong Planning A* (LPA*) improves the performance of A* in a dynamically changing environment by taking into account neighboring nodes when processing the current node being traversed. More specifically, every node n has a predecessor n' from which it is extended, and the node is considered locally consistent if g(n) equals rhs(n). The value rhs(n) is defined as g(n')+d(n',n), where d(n',n) yields the cost of getting from n' to n. Nodes are added to a priority queue for reevaluation when they are locally inconsistent and keyed by two values: first, $min\{g(n), rhs(n)\} + h(n)$, and second, $min\{g(n), rhs(n)\}$. If rhs(n) is less than g(n) (locally overconsistent, the parent n' is now more cheaply reachable), g(n) is set to rhs(n), and if rhs(n) is greater than g(n) (locally underconsistent, the node n is more costly to be reached from n' than previously determined), g(n) is set to infinity. After this, if the node is locally consistent, we pop it from the queue; otherwise, we update its key and add it back to the queue. Since updating the g-value of a node may also affect the rhs-value of the node's successors, all of the node's successors are also reevaluated, leading to a cascading effect as new obstacles are discovered. LPA* essentially allows only a few of the nodes to be expanded again every time an obstacle is encountered rather than all the nodes of the A* graph, which is more efficient.

D* Lite, also described by Koenig and Likhachev [2], is an extension of LPA* which adds a couple of significant optimizations, most notably keying the priority queue for reevaluating nodes differently. Rather than reordering the entire queue, the difference in h(n, goal) between before and after the obstacle was detected is added to every element since the change in the heuristic of every element in the reevaluation queue will be lower bounded by this value. Another optimization is that D* Lite is run from the goal node to the start node since obstacle detection happens closer to the robot rather than the goal node, so we preserve the parts of the tree that are closer to the goal node.

C. RRT^x for Partially Observable Environments

RRT^x, introduced by Otte and Frazzoli [3], uses a priority queue to update weights in a similar fashion to D* Lite, except applied to RRT* rather than A*. In contrast with other RRT* variants, to tackle partially observable environments that may dynamically change, RRT^x has been shown to be asymptotically optimal as well as performing both accurately and quickly.

D. Artificial Potential Fields

Li et al proposed PQ-RRT* in 2020 [4], which builds on RRT* by slightly expanding the search space for finding the nearest node, optimizing the rewiring procedure, and, most importantly, utilizing an artificial potential field to push selected nodes towards the goal state during tree extension.

Also, a repulsive force emanates from obstacles, which further improves the path that the tree follows. Random sampling helps the tree avoid local minima in the field.

III. Methodology

A. Baseline RRT^x

RRT^x is the state-of-the-art approach that was utilized as the baseline since it is both fast in practice and guarantees asymptotic optimality. RRT^x puts all severed nodes into a priority queue based on their cost and rewires each one of them back into the tree. The RRT^x algorithm is shown in figures 1 through 8. Algorithm 1 shows how the $RRT*$-like tree is constructed, where the $shrinkingBallRadius$ algorithm keeps guarantees that the algorithm looks at an average of $log(|V|)$ nodes at each step.

Algorithm 2 shows how points are connected between the existing tree to a new sampled node, as well as creating the permanent neighbor sets and running neighbor sets of the new node. The outgoing permanent neighbors are denoted by N_0^+, the incoming permanent neighbors are denoted by N_0^-, the outgoing running neighbors are denoted by N_r^+, and the incoming running neighbors are denoted by N_r^-. The $findParent$ function in Algorithm 2 finds the most optimal parent for the new node within the nearby neighbor vertex set.

Algorithm 3 looks for new obstacles and propagates the changes to its children while also updating the priority queue with nodes that have inconsistent cost-to-reach-goal,$g(v)$, and lookahead estimates, $lmc(v)$.

Algorithm 4 essentially just rewires neighbors exactly like $RRT*$ with the exception that the priority queue is also updated with inconsistent nodes.

Algorithm 5 propagates the inconsistent cost-to-goal information caused by obstacles with ϵ-consistency, as long as the path that v_{bot} has to traverse is affected; this is the essence of the advantage that RRT^x provides since these updates are fast and effective.

Algorithm 6 ensures that nodes that are added to the orphan set by new obstacles propagate information about the obstacle being added to their descendants, keeping the robot away from obstacles.

Algorithm 7 simply maintains the running neighbors based on the provided shrinking radius described earlier.

Algorithm 8 adds vertices to the orphan set in order to propagate obstacle locations later on.

B. FA-RRT*

We propose a few changes to the existing state-of-the-art solutions for RRT^x with FA-RRT* in partially observable environments. Our solution will instead handle obstacle discovery by first determining some set of tree connections that must be severed, performing a potential field update on inconsistent nodes from the priority queue, and rewiring using the same mechanism as RRT^x.

Algorithm 1 RRT^x

```
1: procedure $RRT^x(obstacles, world)$
2:     $V \leftarrow \{v_{goal}\}$
3:     $v_{bot} \leftarrow v_{start}$
4:     while $v_{bot} \neq v_{goal}$ do
5:         $r \leftarrow shrinkingBallRadius(|V|)$
6:         $updateObstacles(obstacles)$
7:         $v_{bot} \leftarrow updateRobot(v_{bot})$
8:         $v \leftarrow randomNode(world)$
9:         $v_{nearest} \leftarrow nearest(v)$
10:        $steer(v, v_{nearest})$
11:        if $v \cap obstacles = \emptyset$ then
12:            $extend(v, r)$
13:        end if
14:        if $v \in V$ then
15:            $rewireNeighbors(v)$
16:            $reduceInconsistency()$
17:        end if
18:    end while
19: end procedure
```

Algorithm 2 extend

```
1: procedure EXTEND$(v, r)$
2:     $V_{nearby} \leftarrow nearby(v, r)$
3:     $p = findParent(v, V_{near})$
4:     if $p = \emptyset$ then return
5:     end if
6:     $V \leftarrow V \cup \{v\}$
7:     $assignChildParent(p, v)$
8:     for all $u \in V_{near}$ do
9:         if $path(v, u) \cap obstacles = \emptyset$ then
10:            $N_0^+(v) \leftarrow N_0^+(v) \cup \{u\}$
11:            $N_r^-(u) \leftarrow N_r^-(u) \cup \{v\}$
12:        end if
13:        if $path(u, v) \cap obstacles = \emptyset$ then
14:            $N_0^-(v) \leftarrow N_0^-(v) \cup \{u\}$
15:            $N_r^+(u) \leftarrow N_0^+(u) \cup \{v\}$
16:        end if
17:    end for
18: end procedure
```

Algorithm 3 updateObstacles

```
1: procedure UPDATEOBSTACLES$(obstacles)$
2:     if $obstacle_{new} \in obstacles$ then
3:         $addObstacle(obstacle_{new})$
4:     end if
5:     $propagateDescendants()$
6:     $updateQ(v_{bot})$
7:     $reduceInconsistency()$
8: end procedure
```

Algorithm 4 rewireNeighbors

```
procedure REWIRENEIGHBORS($v$)
    if $g(v) - lmc(v) > \epsilon$ then
        $cullNeighbors(v, r)$
        for all $u \in N^{-}(v) - \{v.parent\}$ do
            if $lmc(u) > d(u, v) + lmc(v)$ then
                $lmc(u) \leftarrow d(u, v) + lmc(v)$
                $u.parent = v$
                if $g(u) - lmc(u) > \epsilon$ then
                    $updateQ(u)$
                end if
            end if
        end for
    end if
end procedure
```

Algorithm 5 reduceInconsistency

```
procedure REDUCEINCONSISTENCY()
    while $size(Q) > 0$ and $(key(top(Q)) < key(v_{bot})$ or $lmc(v_{bot}) \neq g(v_{bot})$ or $g(v_{bot} = \infty$ or $v_{bot} \in Q)$ do
        $v \leftarrow pop(Q)$
        if $g(v) - lmc(v) > \epsilon$ then
            $cullNeighbors(v, r)$
            for all $u \in N^{+}(v) - V_{orphan}$ do
                if $lmc(v) > d(v, u) + lmc(u)$ then
                    $p' = u$
                end if
            end for
            $v.parent \leftarrow p'$
            $rewireNeighbors(v)$
        end if
        $g(v) \leftarrow lmc(v)$
    end while
end procedure
```

Algorithm 6 propagateDescendants

```
procedure PROPAGATEDESCENDANTS($v, r$)
    for all $v \in V_{orphan}$ do
        $V_{orphan} \leftarrow V_{orphan} \cup v.children$
    end for
    for all $v \in V_{orphan}$ do
        for all $u \in (N^{+}(v) \cup v.parent) - V_{orphan}$ do
            $g(u) \leftarrow \infty$
            $updateQ(u)$
        end for
    end for
    for all $v \in V_{orphan}$ do
        $g(v) \leftarrow \infty$
        $lmc(v) \leftarrow \infty$
        if $v.parent \neq \emptyset$ then
            $removeParent(v)$
        end if
    end for
    $V_{orphan} \leftarrow \emptyset$
end procedure
```

Algorithm 7 cullNeighbors

```
procedure CULLNEIGHBORS($v, r$)
    for all $u \in N_r^{+}(v)$ do
        if $r < d(v, u)$ and $v.parent \neq u$ then
            $N_r^{+}(v) \leftarrow N_r^{+}(v) - \{u\}$
            $N_r^{-}(u) \leftarrow N_r^{-}(u) - \{v\}$
        end if
    end for
end procedure
```

Algorithm 8 updateQ

```
procedure UPDATEQ($v$)
    if $v \in Q$ then
        $key \leftarrow (min(g(v), lmc(v)), g(v))$
        $updateKey(v, key)$
    end if
end procedure
```

1) Tree-connection Severing: Whenever an obstacle is found, all nodes that fall within the bounds of the obstacle can simply be deleted from the tree. For each of these deleted nodes, maintain a set of the closest ancestors (closer to the goal than self) that are not within the obstacle. Note that many deleted nodes may share the same most recent parent outside of the obstacle. This subset of nodes can also be thought of as the most-child nodes that both have children which were affected by the obstacle but they themselves were not deleted by the obstacle.

Every child (direct and indirect) of any node in this set should be severed from its parent and added to a free set. Note that these children are not being deleted, just severed from the tree. This will result in all children that were broken by the obstacle being free nodes.

2) Potential Field Update: When the algorithm starts, the entire environment is initialized with 0 field force in any direction. Whenever obstacles are detected, the potential field is updated based on the new obstacles. This field update will be done in a very similar way to what is done in the potential field path planning algorithm described above. Then, every

Algorithm 9 addObstacle

```
procedure ADDOBSTACLE($obstacle_{new}$)
    $obstacles \leftarrow obstacles \cup obstacle_{new}$
    for all $(v, u) \in E : (v, u) \cap obstacles\emptyset$ do
        $d(v, u) \leftarrow \infty$
        if $v.parent = u$ then
            if $v \in Q$ then
                $Q.remove(v)$
            end if
            $V_{orphan} \leftarrow V_{orphan} \cup \{v\}$
        end if
    end for
end procedure
```

time inconsistency is reduced using the priority queue, the field is applied to the inconsistent node popped from the queue and a new node is generated by stepping from the old node's most optimal neighbor to the node pushed by the field.

3) Randomized Rewiring Step: After the nodes are pushed by the potential field update, the tree must be rewired. The rewiring happens similar to RRT^x, where nodes are selected from the priority queue based on the lowest cost-to-goal or lookahead estimate. However, since these points have now been pushed by the potential field, the tree steps towards the nodes popped from the queue rather than trying to connect these points directly, as described in the previous subsection.

As in normal RRT, the current robot position will be occasionally returned with some probability. Note that the remaining nodes in the free set are not deleted, so they can still be reused if the robot eventually needs to traverse back around through those nodes. Algorithms 10 through 13 depict the new procedures necessary for FA-RRT*.

C. FA-RRT Pseudocode*

Algorithm 10 provides a basis for FA-RRT* and is very similar to how RRT^x is structured other than two key changes - the $updateObstaclesAndField()$ function replaces $updateObstacles()$ and $reduceInconsistencyWithField()$ replaces $reduceInconsistency()$.

Algorithm 11 is similar to $updateObstacles()$ other than how obstacles are used to create the potential field that $reduceInconsistencyWithField()$ uses.

Algorithm 12 creates a field that is the gradient of a Gaussian Filter on obstacle locations. This is then multiplied by an amount inversely proportional to the fraction of explored space occupied by discovered obstacles. This constant is denoted as k_{pf}.

Algorithm 13 uses the created potential field and applies it to an inconsistent node popped from the priority queue while the loop condition isn't satisfied. It then creates a new node that is stepped toward from the field-updated inconsistent node's parent, preserving optimality of the algorithm. This new node is created and removed from the queue to be used later.

D. Simulation

The simulation used to test the nodes consisted of two maps. The first map has a narrow opening and a box open on one side, while the second map is a very cluttered maze-like environment. The maps are show in the order described in the results section. These maps were used to show how inconsistencies need to propagate as well as how FA-RRT* should avoid dead-ends better.

IV. Results

A. Asymptotic Optimality

Otte and Frazzoli show that RRT^x is asymptotically optimal since it inherits the cost function and ball parameter of $RRT*$ [3]. Since FA-RRT* performs the same obstacle handling procedure with the exception that there is a potential

Algorithm 10 $FA - RRT^*$

```
1: procedure $RRT^x(obstacles, world)$
2:     $V \leftarrow \{v_{goal}\}$
3:     $v_{bot} \leftarrow v_{start}$
4:     while $v_{bot} \neq v_{goal}$ do
5:         $r \leftarrow shrinkingBallRadius(|V|)$
6:         $field \leftarrow updateObstaclesAndField(obstacles)$
7:         $v_{bot} \leftarrow updateRobot(v_{bot})$
8:         $v \leftarrow randomNode(world)$
9:         $v_{nearest} \leftarrow nearest(v)$
10:        $steer(v, v_{nearest})$
11:        if $v \cap obstacles = \emptyset$ then
12:            $extend(v, r)$
13:        end if
14:        if $v \in V$ then
15:            $rewireNeighbors(v)$
16:            $reduceInconsistencyWithField()$
17:        end if
18:    end while
19: end procedure
```

Algorithm 11 updateObstaclesAndField

```
1: procedure UPDATEOBSTACLESANDFIELD($obstacles$)
2:     if $obstacle_{new} \in obstacles$ then
3:         $addObstacle(obstacle_{new})$
4:     end if
5:     $propagateDescendants()$
6:     $updateQ(v_{bot})$
7:     $field \leftarrow createField()$
8:     $reduceInconsistencyWithField(field)$
9:     return $field$
10: end procedure
```

Algorithm 12 createField

```
1: procedure CREATEFIELD()
2:     for all $x, y \in world$ do
3:         if $x, y \in obstacle$ then
4:             $obstacleMask[x, y] \leftarrow 1$
5:         end if
6:     end for
7:     $blur \leftarrow GaussianFilter(obstacleMask, blurSigma)$
8:     $dx, dy = gradient(blur)$
9:     for all $x, y \in world$ do
10:        $potentialField[x, y, 0] += k_{pf} * dx$
11:        $potentialField[x, y, 1] += k_{pf} * dy$
12:    end for
13:    return $potentialField$
14: end procedure
```

Algorithm 13 reduceInconsistencyWithField

```
1:  procedure REDUCEINCONSISTENCYWITHFIELD($field$)
2:      while $size(Q) > 0$ and $(key(top(Q)) < key(v_{bot})$ or $lmc(v_{bot}) \neq g(v_{bot})$ or $g(v_{bot} = \infty$ or $v_{bot} \in Q)$ do
3:          $v \leftarrow pop(Q)$
4:          $v_{field} \leftarrow applyField(v, field)$
5:          if $g(v) - lmc(v) > \epsilon$ then
6:              $cullNeighbors(v, r)$
7:              for all $u \in N^{+}(v_{field}) - V_{orphan}$ do
8:                  if $lmc(v) > d(v, u) + lmc(u)$ then
9:                      $p' = u$
10:                 end if
11:             end for
12:             $v_{new} \leftarrow steer(p', v_{field})$
13:             $v_{new}.parent \leftarrow p'$
14:             $rewireNeighbors(v_{new})$
15:         end if
16:         $g(v) \leftarrow lmc(v)$
17:     end while
18: end procedure
```

field applied, we only need to make sure that the potential field updates and assoicated changes do not affect the optimality.

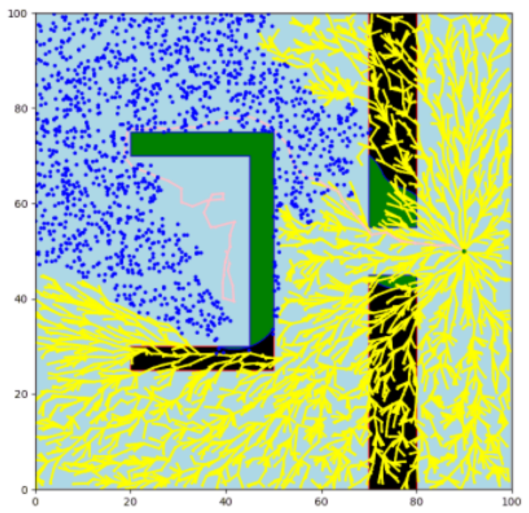
The potential field stays constant in size based on the size of the overall world, which, in a fixed map, is O(1) in runtime and O(A) in memory, where A is the area of the world. The potential field update only occurs on nodes being popped from the priority queue that are inconsistent, which is an O(1) runtime update each time a node is popped. Finally, since the robot is stepping from the inconsistent node's parent towards the field-updated inconsistent node, we ensure that the node stays within a certain radius of the parent and does not break the d-ball optimality. Thus, the FA-RRT* algorithm inherits $\Theta(nlogn)$ runtime from RRT^{x}, where n is the number of vertices in the tree.

B. Information Transfer Time

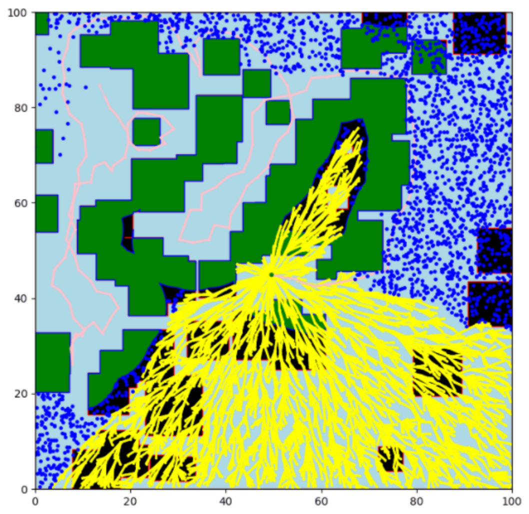
Just like how there is an upper bound to how much a node can be moved to preserve asymptotic optimality, there is a lower bound to the reduction in information transfer time that this implementation of a potential field causes, analytically showing that there is a range in which the improvement lies. At any given point, the potential field affects this point by product of the gradient of a 2D multivariate Gaussian and the inverse of the fraction of area in the observed world occupied by obstacles. The Gaussian gradient term ensures that there is a smooth potential field; however, this field may be two strong or weak given limitations of the environment, not providing a concrete decrease in information transfer time.

Since the factor representing the area that the the obstacles are occupying in the world is included, there is always an ν-improvement in obstacle avoidance, allowing for information about the goal state to be transferred more quickly than RRT^{x}. Just as an ϵ-change in the path due to obstacles is necessary for RRT^{x} to provide an improvement in information transfer time based on inconsistency, a ν-change in the dynamically detected proportion of the observed world that is occupied by obstacles is required for there to be a significant improvement in $FA - RRT$ over RRT^{x} in terms of information transfer time.

C. Evaluation - Baseline:RRT^{x}

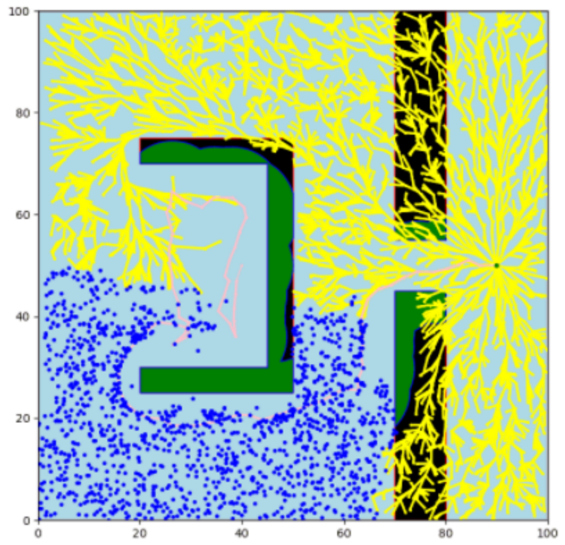


Path length: 161
Timesteps: 61

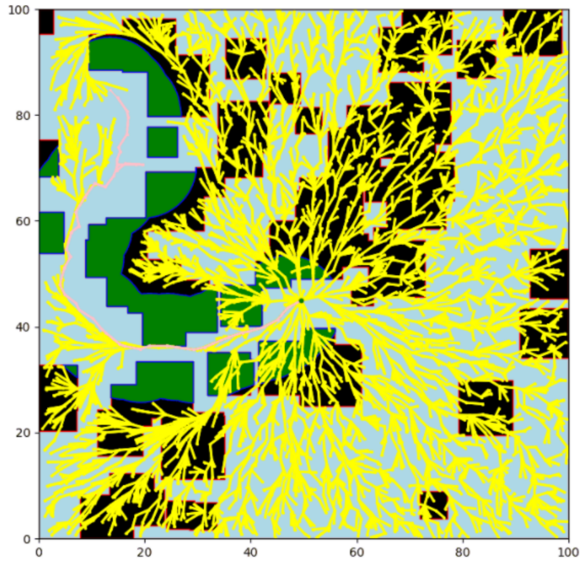


Path length: 392
Timesteps: 144

D. Evaluation - Variation:FA-RRT*



Path length: 216
Timesteps: 84



Path length: 109
Timesteps: 45

V. Discussion

In general, the queue allowed changes to propagate efficiently in FA-RRT* similar to RRT^x. The runtime of O(nlogn) was also preserved, which is a novel addition for using potential fields in conjunction with RRT*. We also showed that decrease in information transfer time has a lower bound with our approach.

In practice, the path length and timesteps required to reach the goal were longer for FA-RRT* in some cases and shorter in others, indicating that actual convergence time was not too different. For the second map in particular, FA-RRT* showed how it can perform significantly better in cluttered environments. This is because RRT^x is somewhat greedy in the way it is always finding the shortest distance to the path, whereas FA-RRT* encodes more information about obstacles occluding the path, particularly dead-ends. The path generated by FA-RRT* is more characteristic of what real-life dynamics of a robot might follow, which makes it more applicable in reality.

A. Limitations

Our study was limited by the quantity and quality of maps simulated since the algorithm needs to generalize well to several maps to prove its practicality. Also, using actual robot dynamics would allow us to show that FA-RRT* is indeed more representative of real robot motion.

References

[1] S. Koenig and M. Likhachev, "Incremental a*," in *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, (Cambridge, MA, USA), p. 1539–1546, MIT Press, 2001.

[2] S. Koenig and M. Likhachev, "D*lite," in *Eighteenth National Conference on Artificial Intelligence*, (USA), p. 476–483, American Association for Artificial Intelligence, 2002.

[3] M. Otte and E. Frazzoli, *RRT^X: Real-Time Motion Planning/Replanning for Environments with Unpredictable Obstacles*, pp. 461–478. 04 2015.

[4] Y. Li, W. Wei, Y. Gao, D. Wang, and Z. Fan, "Pq-rrt*: An improved path planning algorithm for mobile robots," *Expert Systems with Applications*, vol. 152, p. 113425, 2020.