2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

RT-RRT: Reverse Tree Guided Real-Time Path Planning/Replanning in Unpredictable Dynamic Environments

Bo Cui, Rongxin Cui, Weisheng Yan, Yongkang Wang, and Shi Zhang

***Abstract*— Path planning in unpredictable dynamic environments remains a challenging problem due to the unpredictable appearance, disappearance, and movement of dynamic obstacles during navigation. To address this problem, we propose a reverse tree guided rapid exploration random tree (RT-RRT) algorithm that can efficiently perform navigation tasks in dynamic environments. The method first constructs a reverse tree rooted as goal state to search for an initial path. If a collision occurs on the path, The RT-RRT constructs a forward tree rooted as the current robot state in the same configuration space, until it connects with the reverse tree to find a new path. Furthermore, The RT-RRT improves the tree construction method and designs a path optimization strategy to reduce the path cost. The method is validated in different scenarios and has excellent navigation capabilities in unpredictable dynamic environments. In the same scenarios, the RT-RRT algorithm improves the success rate by 16.7%, reduces the path length by 20.54% and reduces the travel time by 10X compared to the RRT^{X} algorithm with the same number of samples.**

I. INTRODUCTION

Online path planning for robots in unpredictable dynamic environments is the process of computing a collision-free path from the robot state to the goal state and repairing it in real time as obstacles change [1]. In most cases, it's possible to replan a path in the new configuration space after detecting obstacles, using the current state of the robot as the start state. However, obstacles may appear, move or disappear unpredictably in dynamic environments with limited perception, unknown a priori environments, high speeds and significant collision risks, such as autonomous underwater vehicle navigation [2]–[4] and high-speed autonomous driving [5]–[7], which pose significant challenges to replanning.

Several algorithms have been proposed for path planning in dynamic environments, the graph-based and sampling-based methods are the most applied [8]–[10]. Graph-based algorithms, such as D* and D*Lite [11], [12], use grid representations of the robot's configuration space, perform cost-based searches, and replan paths. However, these methods rely on map construction and have a low planning efficiency in large environments [13]. Early sampling-based replanning algorithms, such as E-RRT (execution extended RRT), construct search trees with the start state as the root, and reconstruct the search tree with the robot's current position as the root if obstacles change [14]. Algorithms such as GRIP (Greedy, Incremental, Path-directed) [15] and ATS+EB(Adaptive Time-Stepping with Exponential Backoff) [16] replan the path by maintaining a search tree rooted as the starting state. These algorithms are difficult to use for fast planning in dynamic environments because they require significant computation to update the search tree as the robot's state changes [17].

The DRRT (Dynamic Rapidly-exploring Random Trees) algorithm innovatively constructs a reverse tree with the goal state as the root, deletes affected states to repair the reverse tree when the obstacle set is updated, and only needs to maintain a single tree in the configuration space for path replanning [18]. Based on this, the RRT^{X} algorithm designs a reconnection cascade strategy to maintain the reverse tree rooted as the goal state [19]. By storing neighbouring graph connectivity information in each state and updating the subtree iteratively with neighbouring states, it efficiently updates the reverse tree.

Reverse tree performs well in dynamic environments by

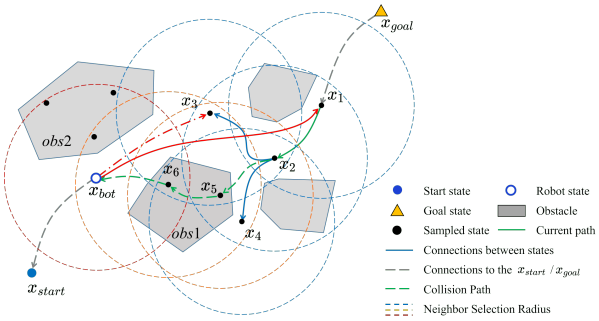


Fig. 1. Replanning schematic diagram for RT-RRT algorithm in $\mathbb{R}^2$ space. A reverse tree is constructed with x_{goal} as the root in the configuration space, generating an initial path. As the robot reaches the state $x_{\text{bot}}(t)$, two new obstacles, $obs1$ and $obs2$, are detected, causing a collision in the current path (green line), and some states in collision, such as x_5 and x_6, need to be discarded. In previous replanning algorithms employing tree repair strategies, such as RRT^{X}, tree repairing is achieved by iteratively rewriting states cascading from existing neighbor states. However, in the depicted scenario, due to the collision of states x_5 and x_6, $x_{\text{bot}}(t)$ is not present in the neighborhood of any existing states, resulting in a high time cost for tree repair. The RT-RRT algorithm doesn't rely on iterative processing of neighbor states. It constructs a forward tree with $x_{\text{bot}}(t)$ as the root, quickly steering to the x_3 state (red dashed line), and further optimizes the connection to x_1 (red solid line) find a path connecting $x_{\text{bot}}(t)$ and x_{goal}. It utilizes the state information of $x_{\text{bot}}(t)$ and x_{goal}, enabling rapid exploration of all possible paths and significant performance improvements.

*This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant U22A2066 and Grant U21B2047, in part by the Key Research and Development Program of Shaanxi under Grant 2022ZDLGY03-05, in part by Innovation Foundation for Doctor Dissertation of Northwestern Polytechnical University under Grant CX2024052 and in part by Shaanxi Innovative Talent Pandeng Program for Young Science and Technology Rising Star Project 2017KJXX-38

Authors are with School of Marine Science and Technology, Northwestern Polytechnical University, Xi'an 710072, China.

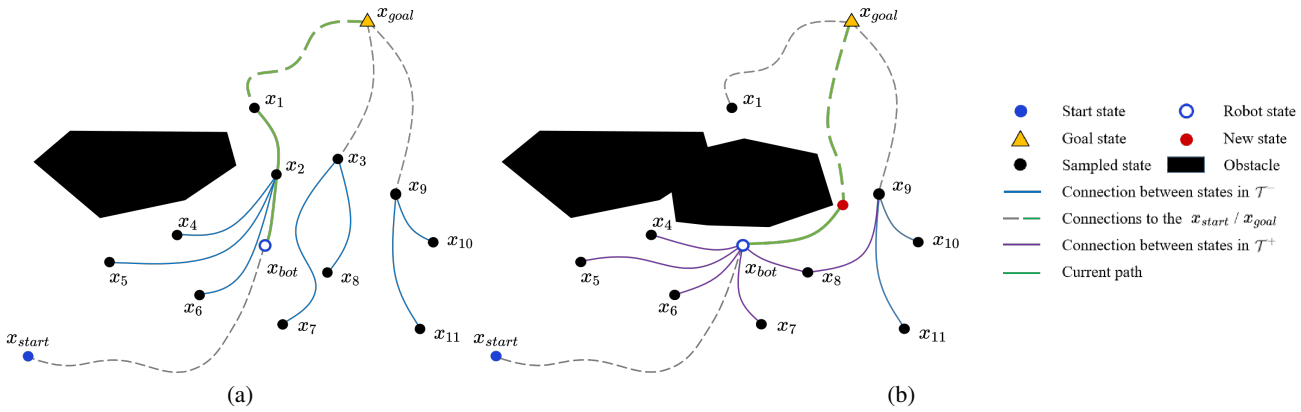


Fig. 2. Schematic representation of the RT-RRT algorithm. (a) is the inverse tree $\mathcal{T}^-$ (blue line) constructed with x_{goal} as the root, and the robot moves along a branch in the tree to the goal (green line). In (b) a dynamic obstacle collides with the original path. The RT-RRT algorithm constructs a tree $\mathcal{T}^+$ with $x_{\text{bot}}(t)$ as root (purple line) and sequentially adds x_4 to x_8 to $\mathcal{T}^+$. Once x_9 has been added to $\mathcal{T}^+$, trees $\mathcal{T}^+$ and $\mathcal{T}^-$ are connected by x_9 and the construction of tree $\mathcal{T}^+$ is terminated. The RT-RRT algorithm then reverses the connection of $x_{\text{bot}}(t) - x_8 - x_9$ in tree $\mathcal{T}^+$ to tree $\mathcal{T}^-$ to repair the tree, while the path optimisation procedure further optimises the path to get the new collision-free path (green line).

simply repairing a same tree without reconstructing it when the environment changes. However, reverse tree based algorithms such as the RRT$^{\text{X}}$ algorithm depend on the graph connectivity of neighbouring states for tree repair. If there are fewer sample states, the states in the reverse tree will have fewer neighbors, making it difficult to keep the path optimal or even failing, If there are many sample states, the repair time of the reverse tree will increase significantly in each iteration.

We propose a reverse tree guided rapidly exploring random tree algorithm, RT-RRT, which constructs both a reserve tree rooted as the goal state and a forward tree rooted as the robot state. Before the robot moves, The RT-RRT constructs a reserve tree, denoted $\mathcal{T}^-$, in the configuration space to obtain the initial path. When the obstacle set is updated as the robot moves, The RT-RRT constructs another tree, $\mathcal{T}^+$, rooted as the current robot state in the same configuration space until a state is found that is connected to both $\mathcal{T}^+$ and $\mathcal{T}^-$. Then the connection of this state in $\mathcal{T}^+$ is reversed and added to $\mathcal{T}^-$ to repair $\mathcal{T}^-$.

An important feature of this algorithm is that it doesn't need to update all the affected states in $\mathcal{T}^-$ when the obstacles are updated, and quickly finds a collision-free path for replanning by constructing a bi-directional tree using the robot state and goal state information. Meanwhile, The RT-RRT discards the use of neighbour states and designs a cost function to sequentially add the sampled states in the configuration space to the tree $\mathcal{T}^+$ when the set of obstacles changes, so that path replanning does not rely on sequential iterations in the neighbour states, which increases the efficiency of planning while improving the success rate (Fig. 1). We evaluated our algorithm in several unpredictable dynamic scenarios and compared it with the RRT$^{\text{X}}$ algorithm. The results show that the RT-RRT algorithm achieves excellent results in terms of both planning time and path length, and can be used for fast path planning in unpredictable dynamic environments.

II. Problem Definition and Terminology

Let $\mathcal{X}$ denote the configuration space. $\mathcal{X}_{\text{obs}} \subset \mathcal{X}$ is the open subset of $\mathcal{X}$ representing the obstacle space, and $\mathcal{X}_{\text{free}} = \mathcal{X} \setminus \mathcal{X}_{\text{obs}}$ is the closed subset representing the free space. The start and goal states of the robot are denoted by x_{start} and x_{goal}, respectively. The state of the robot at time t is denoted by $x_{\text{bot}}(t)$. Let $\mathcal{T}$ denote the directed tree, $\mathcal{T}.\mathcal{V}$ denotes the set of states in the tree $\mathcal{T}$. $\mathcal{T}^+$ denote the directed tree rooted as $x_{\text{bot}}(t)$, and $\mathcal{T}^-$ denote the directed tree rooted as x_{goal}.

The function $\Delta\mathcal{X}_{\text{obs}} = f(t, x_{\text{bot}}(t))$ represents the obstacles detected by the sensors mounted on the robot at time t while the robot is in state $x_{\text{bot}}(t)$.The static environment refers to the case where the function f is known in advance. On the other hand, if the function f is not known in advance and has to be estimated from the robot state $x_{\text{bot}}(t)$ and time t, it is called a dynamic environment. To represent the feasible path of the robot in free space, a continuous mapping is used, denoted as $\sigma : [0, 1] \to \mathcal{X}_{\text{free}}$. When the robot travels from state x_1 to x_2 , where x_1 is the start state and x_2 is the current state (i.e., $0 \mapsto x_1$ and $1 \mapsto x_2$), the trajectory can be represented as $\sigma(x_1, x_2)$ and its length is $d_\sigma(x_1, x_2)$.

We define the "shortest path replanning" problem following the literature [19]. Given the configuration space $\mathcal{X}$, $\mathcal{X}_{\text{free}}$, $\mathcal{X}_{\text{obs}}$, the robot start state x_{start} and the goal state x_{goal}. Compute the optimal path $\sigma^*(x_{\text{bot}}(t), x_{\text{goal}})$ to guide the robot from $x_{\text{bot}}(t_0) = x_{\text{start}}$ to $x_{\text{bot}}(t) = x_{\text{goal}}$. Update $\Delta\mathcal{X}_{\text{obs}}$ when $x_{\text{bot}}(t)$ and time t is updated and recalculate $\sigma^*(x_{\text{bot}}(t), x_{\text{goal}})$ when $\Delta\mathcal{X}_{\text{obs}} \neq \emptyset$, where

$$\sigma^*(x_{\text{bot}}(t), x_{\text{goal}}) = \underset{\sigma(x_{\text{bot}}(t), x_{\text{goal}}) \in \mathcal{X}_{\text{free}}}{\arg\min} d_\sigma(x_{\text{bot}}(t), x_{\text{goal}}) \quad (1)$$

III. The RT-RRT Algorithm

A. Minor Algorithms

Before outlining our approach, we describe the procedures on which the main algorithms depend.

- **SampleFree**(x): a function that returns a random state in the configuration space.
- **Nearest**$(\mathcal{T}.\mathcal{V}, x)$: a function that returns the nearest state to state x in the set $\mathcal{T}.\mathcal{V}$.
- **Steer**(x, y, ε): a function that returns a state that steers a step of ε from state x to state y.
- **Sort**$(\mathcal{T}.\mathcal{V}, cost)$: a function that returns a set that sorts the states in $\mathcal{T}.\mathcal{V}$ by $cost$.
- **SortNearest**$(\mathcal{T}.\mathcal{V}, x)$: a function that calculates the cost of the states on the tree $\mathcal{T}.\mathcal{V}$ to x and returns a set that sorts them from smallest to largest according to cost.
- **SortAncestor**$(\mathcal{T}.\mathcal{V}, x)$: a function that returns the set of parent of x on $\mathcal{T}$, as well as the ancestor of the parent states, and sorts them from the 'oldest' ancestor to $\mathcal{T}$.parent(x).
- **CollsionFree**$(\sigma(x, y), \mathcal{T})$: a function that tests whether the connection from x to y in tree $\mathcal{T}$ belongs to $\mathcal{X}_{\text{free}}$.
- **needsReplanning**$(\sigma(x, y), \mathcal{T})$: a function that determines if any collision occurs on a path $\sigma(x, y)$ in tree $\mathcal{T}$, returning a bool value.

B. Overview

The RT-RRT algorithm is designed in two steps. First, a reverse tree $\mathcal{T}^-$ rooted as x_{goal} is constructed to get the initial path. Then the robot moves. If a collision occurs on the path at time t, a forward tree $\mathcal{T}^+$ rooted as $x_{\text{bot}}(t)$ is constructed in the same configuration space as $\mathcal{T}^-$ until a state is found that is connected to $\mathcal{T}^-$, and this connection is added to $\mathcal{T}^-$ to get a new collision-free path.

Algorithm 1 RT-RRT

1: $\mathcal{T}^-.\mathcal{V} \leftarrow \{x_{\text{goal}}\}$
2: $x_{\text{bot}}(t_0) \leftarrow x_{\text{start}}$
3: CreateNewTree$^-$()
4: **while** $x_{\text{bot}}(t) \neq x_{\text{goal}}$ **do**
5: **if** *obstacles have changed* **then**
6: updateObstacles()
7: **end if**
8: **if** *robot is moving* **then**
9: $x_{\text{bot}}(t) \leftarrow$ updateRobot$(x_{\text{bot}}(t))$
10: **end if**
11: **if** needsReplanning$(\sigma(x_{\text{bot}}(t), x_{\text{goal}}), \mathcal{T}^-)$ **then**
12: CreateNewTree$^+(\mathcal{T}^-, x_{\text{bot}}(t))$
13: **end if**
14: OptimizePath$(\mathcal{T}^-, x_{\text{bot}}(t), D_{\text{Threshold}})$
15: **end while**

The main program is shown in Algorithm 1 and the main subroutines are shown in Algorithms 2-7. Before the robot moves, the $CreateNewTree^-()$ function (line 3) constructs a reverse tree $\mathcal{T}^-$ rooted as x_{goal} to cover the configuration space and get the initial path. Then, the robot moves and the main loop (lines 4-15) ends when the robot reaches the goal. The configuration space is updated whenever obstacles are updated or the robot moves (lines 5-10). If the path from $x_{\text{bot}}(t)$ to x_{goal} in the reverse tree $\mathcal{T}^-$ collides with any obstacles, the $CreateNewTree^+()$ function (line 12) constructs a forward tree $\mathcal{T}^+$ with $x_{\text{bot}}(t)$ as root until a state connecting $\mathcal{T}^+$ and $\mathcal{T}^-$ is found and rewrite to $\mathcal{T}^-$, then optimizes the connection using the $OptimizePath()$ function to get a new path(line 14).

C. Construction of $\mathcal{T}^-$

A reverse tree, denoted $\mathcal{T}^-$, rooted as x_{goal} can be repaired rather than reconstructed as obstacles are updated, making it well suited for dynamic environments. In the construction of tree, the parameter of neighbor selection radius R_{near} has a significant impact on both computation time and path length. While increasing R_{near} can reduce the path cost, it also increases the number of states computed in each iteration, resulting in a significant increase in computation time. We use ancestor states to construct the reverse tree instead of using the neighbor states according to R_{near}, which greatly reduces the path cost and planning time [20], [21].

The algorithm 2 describes the reverse tree construction process. After adding a new state x_{new} to the tree $\mathcal{T}^-$ (lines 1-4), we rewrite the connection using the ancestor states $x_{\text{ancestor}} \in X_{\text{ancestor}}$. By the triangle inequality, the cost from x_{new} to x_{near} to x_{ancestor} is necessarily less than the cost from x_{new} to x_{ancestor}. Consequently, we consider the ancestor states X_{ancestor} of x_{nearest} as potential parent states (line 5) and select the 'oldest' collision-free ancestor state as the parent of x_{new} to expanding $\mathcal{T}^-$ (lines 6-12).

Algorithm 2 CreateNewTree$^-$

1: **for** $i = 1$ **to** n_{samples} **do**
2: $x_{\text{rand}} \leftarrow$ SampleFree(i)
3: $x_{\text{nearest}} \leftarrow$ Nearest$(\mathcal{T}^-.\mathcal{V}, x_{\text{rand}})$
4: $x_{\text{new}} \leftarrow$ Steer$(x_{\text{nearest}}, x_{\text{rand}}, \varepsilon)$
5: $X_{\text{ancestor}} \leftarrow$ SortAncestor$(\mathcal{T}^-.\mathcal{V}, x_{\text{nearest}})$
6: **for each** $x_{\text{ancestor}} \in X_{\text{ancestor}} \cup x_{\text{nearest}}$ **do**
7: **if** CollsionFree$(\sigma(x_{\text{new}}, x_{\text{ancestor}}), \mathcal{T}^-)$ **then**
8: $\mathcal{T}^-.\mathcal{V} \leftarrow \mathcal{T}^-.\mathcal{V} \cup \{x_{\text{new}}\}$
9: $\mathcal{T}^-$.parent$(x_{\text{new}}) \leftarrow x_{\text{ancestor}}$
10: BREAK
11: **end if**
12: **end for**
13: **end for**

D. Construction of $\mathcal{T}^+$

The construction of the tree $\mathcal{T}^+$ is similar to that of the tree $\mathcal{T}^-$. The difference is that $\mathcal{T}^+$ is rooted at $x_{\text{bot}}(t)$, and its expansion doesn't rely on random states, but on the set of states in $\mathcal{T}^-$.

Algorithm 3 outlines the construction of $\mathcal{T}^+$. First, $x_{\text{bot}}(t)$ is added to $\mathcal{T}^+$ (line 1). The cost values corresponding to the state in $\mathcal{T}^-$ are calculated and sorted (line 2). Starting from the states with the smallest cost values, they are sequentially added to $\mathcal{T}^+$ until a state connecting $\mathcal{T}^+$ and $\mathcal{T}^-$ is found, and the connection is rewrite to $\mathcal{T}^-$ (lines 3-17).

Algorithm 4 outlines the method for calculating the costs of states. The cost from a state x to x_{goal} in $\mathcal{T}^-$ is represented

as $ceg(x)$, and the cost to $x_{\text{bot}}(t)$ is represented as $cer(x)$. The cost of a state x is composed of two parts, $ceg(x)$ and $cer(x)$, defined as follows:

$$cost(x) = \begin{cases} cer(x) + ceg(x) & \text{if } cer(x) < R \\ cer(x) + max(ceg(\mathcal{T}^-.\mathcal{V})) & \text{else} \end{cases} \tag{2}$$

Where the parameter R is the Euclidean distance from a state to $x_{\text{bot}}(t)$. When R is set to 0, $max(ceg(\mathcal{T}^-.\mathcal{V}))$ is a constant, and the cost depends only on $ceg(x)$. As R increases, the algorithm tends to construct $\mathcal{T}^+$ in a greedy strategy. However, an excessively large R will result in states with low-cost being distributed around only one single path, even if they are far from $x_{\text{bot}}(t)$ or if that path would lead to collisions, which increasing the construction time of the tree. Therefore, choosing an appropriate value for R balances the consideration of heuristic path finding with the prioritization of selecting states with the shortest distance.

Algorithm 3 CreateNewTree$^+(\mathcal{T}^-.\mathcal{V}, x_{\text{bot}}(t))$

$\mathcal{T}^+.\mathcal{V} \leftarrow \{x_{\text{bot}}(t)\}$
$X_{sort} \leftarrow$ CalculateCost$(\mathcal{T}^-.\mathcal{V}, x_{\text{bot}}(t))$
for each $x_{\text{sort}} \in X_{sort}$ **do**
  $x_{\text{nearest}} \leftarrow$ Nearest$(\mathcal{T}^+.\mathcal{V}, x_{\text{sort}})$
  $X_{\text{ancestor}} \leftarrow$ SortAncestor$(\mathcal{T}^+.\mathcal{V}, x_{\text{nearest}})$
  for each $x_{\text{ancestor}} \in X_{\text{ancestor}} \cup x_{\text{nearest}}$ **do**
    if CollsionFree$(\sigma(x_{\text{nearest}}, x_{\text{bot}}(t)), \mathcal{T}^+)$ **then**
      $\mathcal{T}^+.\mathcal{V} \leftarrow \mathcal{T}^+ \cup \{x_{\text{sort}}\}$
      $\mathcal{T}^+$.parent$(x_{\text{sort}}) \leftarrow x_{\text{ancestor}}$
      BREAK
    end if
  end for
  if CollsionFree$(\sigma(x_{\text{sort}}, x_{\text{goal}}), \mathcal{T}^-)$ **then**
    RewritePath$(\mathcal{T}^-, \mathcal{T}^+, x_{\text{sort}})$
    BREAK
  end if
end for

Algorithm 4 CalculateCost$(\mathcal{T}^-, x_{\text{bot}}(t))$

for x **in** $\mathcal{T}^-.\mathcal{V}$ **do**
  if cer$(x) < R$ **then**
    cost$(x) \leftarrow$ cer(x) + ceg(x))
  else
    cost$(x) \leftarrow$ cer$(x) + max($ceg$(x))$
  end if
end for
$X_{sort} \leftarrow$ Sort$(\mathcal{T}^-.\mathcal{V},$ cost$)$
RETURN X_{sort}

Algorithm 5 outlines the process of rewriting the path. After finding the state x that connects $\mathcal{T}^+$ and $\mathcal{T}^-$, the connection of x in $\mathcal{T}^+$ is reversed and connected to $\mathcal{T}^-$, repairing $\mathcal{T}^-$ and obtaining a new path.

Algorithm 5 RewritePath$(\mathcal{T}^-, \mathcal{T}^+, x)$

$X_{\text{ancestor}} \leftarrow$ SortAncestor$(\mathcal{T}^+.\mathcal{V}, x)$
for x_{i} **in**X_{ancestor} **do**
  if not isEmpty$(x_{\text{i+1}})$ **then**
    $\mathcal{T}^-$.parent$(x_{\text{i}}) \leftarrow x_{\text{i+1}}$
  else
    $\mathcal{T}^-$.parent$(x_{\text{i}}) \leftarrow x$
  end if
end for

E. Optimize Path

Algorithm 6 outlines the path optimization procedure where, at each iteration, the parent state $x^{(2)}$ of $x^{(1)}$ and the parent state $x^{(3)}$ of $x^{(2)}$ are obtained first. Then, by invoking the $FindOptimalState()$ function of Algorithm 7, the optimal state, denoted x_{new}, is determined by dichotomy and used in the subsequent iteration to derive $x_{\text{optimalState}}$. After getting $x_{\text{optimalState}}$, it is integrated into the directed tree $\mathcal{T}^-$ as a new parent of state x and a new child of state $x^{(3)}$.

Algorithm 6 OptimizePath $(\mathcal{T}^-, D_{\text{Threshold}})$

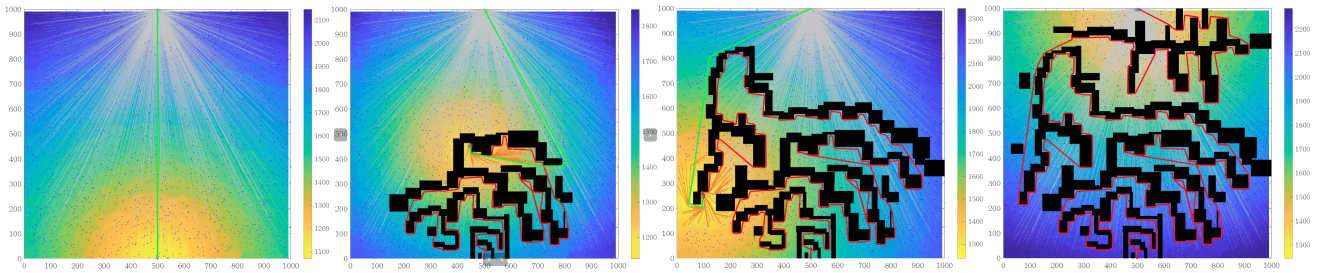
$x^{(1)} \leftarrow x_{\text{bot}}(t)$
while $x^{(2)} \neq x_{\text{goal}}$ **do**
  $x^{(2)} \leftarrow \mathcal{T}^-$.parent$(x^{(1)})$
  $x^{(3)} \leftarrow \mathcal{T}^-$.parent$(x^{(2)})$
  $x_{\text{new}} \leftarrow$ FindOptimalState$(x^{(1)}, x^{(2)}, x^{(3)}, D_{\text{Threshold}})$
  $x_{\text{optimalState}} \leftarrow$ FindOptimalState$(x^{(3)}, x_{\text{new}}, x^{(1)}, D_{\text{Threshold}})$
  $\mathcal{T}^+.\mathcal{V} \leftarrow x_{\text{optimalState}}$
  $\mathcal{T}^-$.parent$(x^{(1)}) \leftarrow x_{\text{optimalState}}$
  $\mathcal{T}^-$.parent$(x_{\text{optimalState}}) \leftarrow x^{(3)}$
  $x^{(1)} \leftarrow \mathcal{T}^-$.parent$(x^{(1)})$
end while

Algorithm 7 FindOptimalState $\left(x^{(1)}, x^{(2)}, x^{(3)}, D_{\text{Threshold}}\right)$

while $d\left(x^{(2)}, x^{(3)}\right) < D_{\text{Threshold}}$ **do**
  $x_{\text{mid}} \leftarrow (x^{(2)} + x^{(3)})/2$
  if CollsionFree$(x^{(1)}, x_{\text{mid}})$ **then**
    $x^{(3)} \leftarrow x_{\text{mid}}$
  else
    $x^{(2)} \leftarrow x_{\text{mid}}$
  end if
  $x_{\text{optimalState}} \leftarrow x^{(2)}$
  return $x_{\text{optimalState}}$
end while

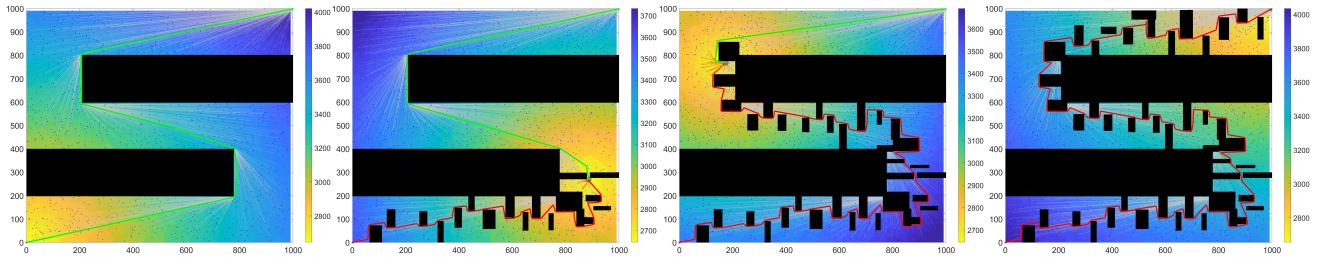
IV. Simulations and Experiments

In subsection A of this section, we show the application of the RT-RRT algorithm in various dynamic environments. In subsection B, we perform a series of experiments using the RT-RRT and RRT$^{\text{X}}$ algorithms in dynamic environments



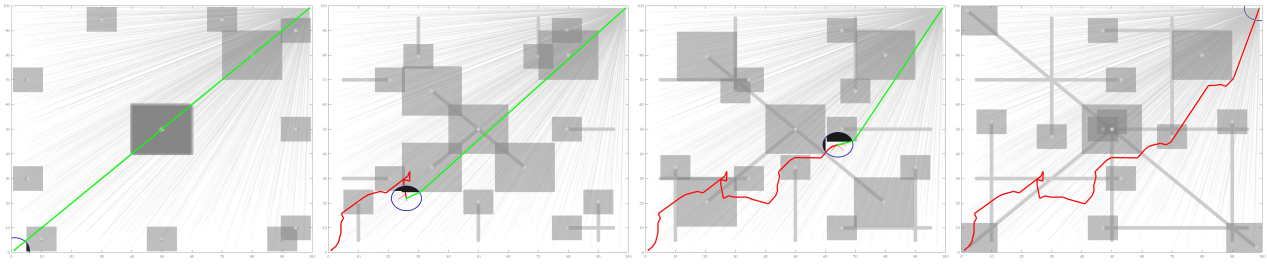
(a) step 1 (b) step 804 (c) step 1431 (d) step 2101

Fig. 3. Robot moving from x_{start} to x_{goal} in a maze scenario using RT-RRT (trajectory in red) while constructing directed trees (gray and yellow) to find the shortest path (green). Obstacles are shown in black. Colored areas are state costs.



(a) step 1 (b) step 309 (c) step 630 (d) step 891

Fig. 4. Robot moves from x_{start} to x_{goal} using RT-RRT in the channel scenario, with both known static obstacles (two black regions in (a)) and dynamic obstacles that appear as the robot moves.



(a) step 1 (b) step 28 (c) step 31 (d) step 92

Fig. 5. Robot moving from x_{start} to x_{goal} using RT-RRT in the unpredictable moving obstacles scenario. The positions of all obstacles (grey rectangles) are unknown a priori to the robot. The obstacles are moving in space, and the robot relies only on sensor (blue circle) to detect obstacles (Marked the detected obstacles as black areas) in real time.

to compare the performance of the RT-RRT with the RRT$^{\text{X}}$. We chose the RRT$^{\text{X}}$ algorithm because it is also a single-query sampling-based path planning algorithm that plans path using a reverse tree. In subsection C, we conducted a experiment based on ROS in a simulation environment using a UAV(unmanned aerial vehicle).

A. Simulations in Dynamic Environments

We assume that the robot is a point-robot, simplifying to the case of geometric path-planning, $\sigma(x, y)$ is a straight line between x and y. The sensor model is simplified to a circle. The dynamic obstacles detected in the sensor range are considered as known obstacles. Some examples of dynamic environments are:

- Obstacles that appear and disappear randomly over time.
- Position of obstacle changes over time.
- Obstacles 'appear' or 'disappear' depending on whether the sensors mounted in robot detect them or not.

Fig. 3-5 show the travel of the robot in different scenarios. The detection range of the robot is represented by a blue circle, the center of which is the robot's position. The green line is the current collision-free path planned by the RT-RRT algorithm, and the red line is the trajectory that the robot traveled. The black and gray rectangles are obstacles. The gray line is the reverse tree $\mathcal{T}^-$ constructed by the RT-RRT algorithm, the yellow line is the tree $\mathcal{T}^+$, and the blue points are the sampled states. colormap is the cost of states.

1) Maze Scenario: A large-scale unknown maze scenario of $1000m \times 1000m$ where obstacles are detected as the robot moves. The robot has an on-board sensor with a detection radius of 10m (blue circle) and moves at a speed of $5m/step$. the start state is located at $(500m, 0m)$ and the goal state is

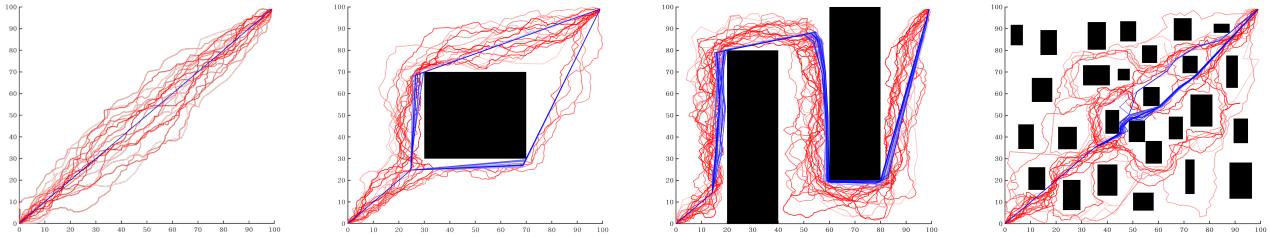
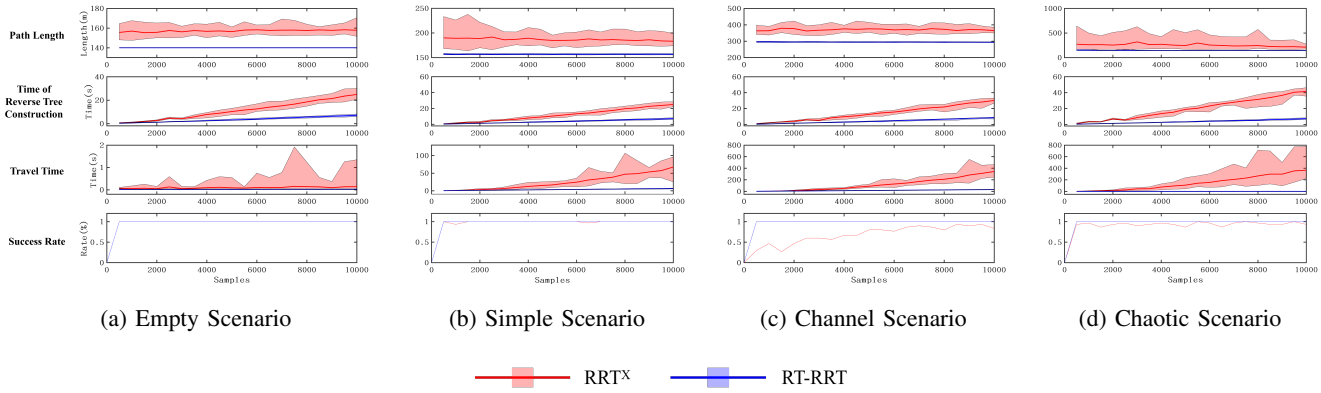


Fig. 6. Paths traveled by the robot by the RT-RRT algorithm (blue line) vs. the RRT^{X} algorithm (red line) for different numbers of samples. The color of the path becomes 'darker' as the number of samples increases.



(a) Empty Scenario (b) Simple Scenario (c) Channel Scenario (d) Chaotic Scenario

Fig. 7. The performance statistics show the path length, time of reverse tree construction, travel time, and success rate of the RT-RRT algorithm (blue) and the RRT^{X} algorithm (red) in different environments with different numbers of sample states.

located at $(500m, 1000m)$.The RT-RRT algorithm constructs a reverse tree $\mathcal{T}^{-}$ using only 2000 sampled states to get the initial path and replans the path after detecting an obstacle until the robot reaches the goal safely(Fig. 3).

2) Channel Scenario: A large-scale channel scenario of $1000m \times 1000m$ with two rectangles of known static obstacles and unknown obstacles not detected in the scenario. The robot has an on-board sensor with a detection radius of $10m$ (blue circle) and moves at a speed of $5m/step$. the start state is located at $(0m, 0m)$, and the goal state is located at $(999m, 999m)$. the construction of the reverse tree in the RT-RRT algorithm also uses only 2,000 sampled states, and navigates the robot to reach the goal safely, even in a narrow region(e.g., Fig. 4(b)).

3) Moving Obstacles Scenario: A moving obstacle scenario of $100m \times 100m$ with both unpredictable moving obstacles and static obstacles in the scenario. The robot has an on-board sensor with a detection radius of $5m$ (blue circle) and moves at a speed of $2m/step$. The obstacles get a movement speed of $0.5m/step$ in x-axis or y-axis. the movement and position of the obstacles are unknown to the robot, and the robot only senses the environment in real time through the sensor. The start state is located at $(0m, 0m)$ and the goal state is located at $(99m, 99m)$. The RT-RRT algorithm uses 1000 sampled states to construct a reverse tree that safely navigates the robot to the goal state in this scenario with moving obstacles (Fig. 5).

B. Experimental Results

In this subsection, we compare the RT-RRT algorithm with the RRT^{X} algorithm [19]. The tests were performed in four different scenarios, all $100m \times 100m$ in size, including empty, simple, channel, and chaotic scenarios (Fig. 6). The obstacles in these scenarios are all unknown and are detected as the robot moves. The robot starts at $(0m, 0m)$ and reaches its goal at $(99m, 99m)$. The robot moves at a speed of $5m/step$, and the detection radius of the on-board sensors is $10m$. For the RRT^{X} algorithm, we tested it based on the open source [22] and set its *ball constant* to $100m$, same as its default value. The R in the RT-RRT algorithm is set to $60m$, and $D_{\text{Threshold}}$ is set to $5m$, The probability of selecting x_{start} during sampling is set to 0.1.

Both the RRT^{X} and RT-RRT algorithms implement dynamic environmental path planning by constructing and repairing a reverse tree, therefore the construction of the reverse tree is significant. The initial number of samples in the reverse tree is an important parameter, where more samples reduce the path cost and improve the planning success rate, but also increase the time to repair the tree in each iteration. Fewer samples may reduce the travel time but result in paths that are difficult to keep optimal. We constructed the directed tree with different numbers of samples. The steer length of the reverse tree for different

TABLE I

STATISTICS ON THE AVERAGE VALUE FOR EACH SAMPLE NUMBER. TIME_0 IS TAKEN AS THE TIME OF REVERSE TREE CONSTRUCTION AND TIME_1 IS TAKEN AS THE ROBOT'S TRAVEL TIME.

Scenario	Algorithm	Path Length(m)			Time_0(s)		Time_1(s)		Success Rate
		min	max	mean	min	max	min	max	mean
Empty	RRT^{X}	155.3449	158.3053	157.2303	0.7231	24.9032	0.0519	0.1545	1.0000
	RT-RRT	140.0071	140.0071	140.0071	0.3102	7.1252	0.0156	0.0213	1.0000
Simple	RRT^{X}	183.2125	191.7116	186.8581	0.7383	24.9885	0.4242	67.6202	0.9950
	RT-RRT	156.3116	157.0058	156.7702	0.3144	7.2743	0.3661	6.7695	1.0000
Channel	RRT^{X}	362.5794	379.2370	370.5190	0.9612	30.1429	1.7030	344.1641	0.7000
	RT-RRT	293.1988	296.4645	294.4303	0.2782	8.5129	1.7408	31.6655	1.0000
Chaotic	RRT^{X}	211.7207	319.3194	253.7875	0.9909	41.4583	1.3652	366.7647	0.9417
	RT-RRT	143.0377	148.1523	144.2837	0.3203	7.3213	0.2116	0.8331	1.0000

numbers of samples is given:

$$\varepsilon = \frac{L_{max}}{\sqrt{N_{max}}}$$

Where L_{max} is the maximum length in space, i.e., the diagonal length, and N_{max} is the number of samples.

We increased the number of samples from 0 at intervals of 500 to 10,000 in four scenarios, and performed 30 validations of the RT-RRT and RRT^{X} algorithms at each number of samples, respectively, for a total of 4,800 experiments evaluated. The path length, time of reverse tree construction, travel time, and success rate were counted separately.

The performance statistics of the RRT^{X} and RT-RRT algorithms for different scenarios are shown in Figure 7. The red and blue lines represent the average value of the performance statistics at each sample number, while the upper and lower bounds of the red and blue regions represent the maximum and minimum values of the statistics at each sample number, respectively. For the RRT^{X} and RT-RRT algorithms, the path length slowly decreases as the number of samples increases (e.g., Fig. 7(d)), and both the reverse tree construction time and the travel time increase. While, the success rate of the RRT^{X} algorithm gradually increases (e.g., Fig. 7.(c)).

In particular, the RT-RRT algorithm grows and repairs the search tree significantly faster than the RRT^{X} algorithm, which is as expected. On the one hand, in RRTx, the higher the number of samples, the more states are affected by new obstacles and the more states need to be reconnected into the reverse tree. On the other hand, as the number of samples increases, the RRT^{X} algorithm relies on information transfer between neighbouring states, and an increase in the number of samples leads to a rapid increase in the number of neighbours per state, which increases the success rate of path planning and reduces the path cost. However, it also leads to a rapid increase in tree construction and repair time.

The RT-RRT algorithm does not rely on neighbour states and can construct the tree $\mathcal{T}^+$ by connecting all possible states to $x_{\text{bot}}(t)$. The repair of $\mathcal{T}^-$ can be achieved by simply finding one state that connects $\mathcal{T}^-$ and $\mathcal{T}^+$ and adding the branches in the tree $\mathcal{T}^+$ to $\mathcal{T}^-$. The path optimisation procedure further reduces the cost of the new paths. In contrast to the RRT^{X} algorithm, the RT-RRT algorithm does not need to repair all states affected by obstacles and their sub-states whenever obstacles are updated. It uses the goal state x_{goal} and the current state $x_{\text{bot}}(t)$ of the robot to improve planning efficiency. The bi-directional tree construction also improves the success rate of the RT-RRT algorithm compared to the uni-directional tree constructed by RRT^{X}.

We also counted the maximum and minimum values of the mean line of each statistic based on the average results under different sample numbers, and the specific results are shown in Table 1 (we use Time_0 to denote the time of reverse tree construction, and Time_1 to denote the travel time of the robot). Analysing the data in Table 1, the average values of the path lengths of the RT-RRT algorithm in empty, simple, channel and chaotic scenes are reduced by 11.00%, 16.10%, 20.54% and 43.15%, respectively, compared to RRT^{X}. The maximum time of Time_0 was reduced by 74.10% on average. The travel time was reduced by 10X (in simple and channel scenarios) and up to 99.9%(due to the robot getting 'lost' by the RRT^{X} in the chaotic scenario). The success rate was increased up to 30.00%(in the channel scenario).

We select the channel scenario that have similar paths traveled by the robots for analysis. Taking an average of the data in 10,000 samples, we can conclude that the RT-RRT algorithm improves the success rate by 16.7%, reduces the path length by 20.54%, and reduces the travel time by 10X compared to the RRT^{X} algorithm. These results show that the RT-RRT algorithm achieves a significant advantage over the RRT^{X} algorithm in terms of both efficiency and quality of path planning.

C. Simulation in Gazebo

We built a simulation environment in Gazebo based on the open source platform Prometheus [23], which provides a UAV equipped with binocular camera, depth camera, lidar, IMU and GPS for real-time mapping (ORB SLAM) and path tracking control. The simulation scenario in Gazebo is shown in Fig. 8.

We have randomly distributed obstacles of different sizes in Gazebo, the location of which is unknown. The starting state of the UAV is $(-10m, 0m, 1m)$ and the goal is set to $(15m, 0m, 1m)$. Fig. 9 visualises the experimental data. Throughout the movement of the UAV, the sensors map the environment in real time based on ORB SLAM. Meanwhile,

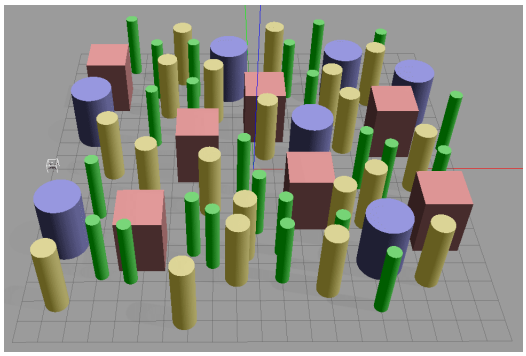


Fig. 8. The scenario in gazebo. The UAV (marked by the white box) must map the environment and replan its path in real time to avoid obstacles (colored cylinders and squares) until it reaches the goal.

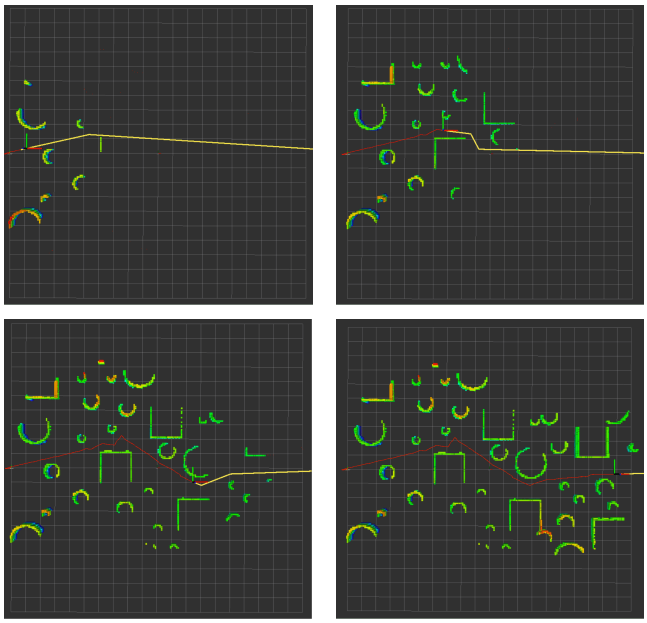


Fig. 9. Data visualisation in RVIZ. The UAV relies on the sensors to map the environment in real time (coloured lines), and the RT-RRT algorithm plans a path in real time (yellow lines) to guide the UAV to the goal (red lines represent the trajectory).

the RT-RRT algorithm plans the path in real time to ensure that the UAV reaches the goal safely.

V. CONCLUSIONS

We propose the RT-RRT algorithm for efficient real-time path planning/replanning in unpredictable dynamic environments. The RT-RRT constructs a reverse tree before navigation and constructs a forward tree in the same configuration space when obstacles set is update during navigation until it is connected to the reverse tree, and adds this connection to the reverse tree to repair the tree. Simulations and comparative experiments with the RRT^X algorithm have been carried out in various scenarios. The simulation and experimental results show the effectiveness of our approach in dynamic environments. In the future, we will apply the RT-RRT algorithm to a real robot to validate the algorithm in the real world.

REFERENCES

[1] Hou B, Srinivasa S S. Dynamic replanning with posterior sampling[C]//2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2022: 2938-2945.

[2] Duecker D A, Horst C, Kreuzer E. From Aerobatics to Hydrobatics: Agile Trajectory Planning and Tracking for Micro Underwater Robots[C]//2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2021: 8617-8624.

[3] Willners J S, Toohey L, Petillot Y. Sampling-based path planning for cooperative autonomous maritime vehicles to reduce uncertainty in range-only localization[J]. IEEE Robotics and Automation Letters, 2019, 4(4): 3987-3994.

[4] Boyer F, Lebastard V, Chevallereau C, et al. Underwater reflex navigation in confined environment based on electric sense[J]. IEEE transactions on robotics, 2013, 29(4): 945-956.

[5] Lei L, Luo R, Zheng R, et al. KB-Tree: Learnable and Continuous Monte-Carlo Tree Search for Autonomous Driving Planning[C]//2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2021: 4493-4500.

[6] Chen Y, Xin R, Cheng J, et al. Efficient speed planning for autonomous driving in dynamic environment with interaction point model[J]. IEEE Robotics and Automation Letters, 2022, 7(4): 11839-11846.

[7] Ding W, Zhang L, Chen J, et al. Epsilon: An efficient planning system for automated vehicles in highly interactive environments[J]. IEEE Transactions on Robotics, 2021, 38(2): 1118-1138.

[8] Sintov A, Borum A, Bretl T. Motion planning of fully actuated closed kinematic chains with revolute joints: A comparative analysis[J]. IEEE Robotics and Automation Letters, 2018, 3(4): 2886-2893.

[9] Pore A, Li Z, Dall'Alba D, et al. Autonomous Navigation for Robot-Assisted Intraluminal and Endovascular Procedures: A Systematic Review[J]. IEEE Transactions on Robotics, 2023.

[10] Claussmann L, Revilloud M, Gruyer D, et al. A review of motion planning for highway autonomous driving[J]. IEEE Transactions on Intelligent Transportation Systems, 2019, 21(5): 1826-1848.

[11] Stentz A. The focussed d* algorithm for real-time replanning[C]//IJCAI. 1995, 95: 1652-1659.

[12] Koenig S, Likhachev M. D* lite[C]//Eighteenth national conference on Artificial intelligence. 2002: 476-483.

[13] Aggarwal S, Kumar N. Path planning techniques for unmanned aerial vehicles: A review, solutions, and challenges[J]. Computer Communications, 2020, 149: 270-299.

[14] Bruce J, Veloso M. Real-time randomized path planning for robot navigation[C]//IEEE/RSJ international conference on intelligent robots and systems. IEEE, 2002, 3: 2383-2388.

[15] Bekris K E, Kavraki L E. Greedy but safe replanning under kinodynamic constraints[C]//Proceedings 2007 IEEE International Conference on Robotics and Automation. IEEE, 2007: 704-710.

[16] Hauser K. On responsiveness, safety, and completeness in real-time motion planning[J]. Autonomous Robots, 2012, 32: 35-48.

[17] Rubio F, Valero F, Llopis-Albert C. A review of mobile robots: Concepts, methods, theoretical framework, and applications[J]. International Journal of Advanced Robotic Systems, 2019, 16(2): 1729881419839596.

[18] Ferguson D, Kalra N, Stentz A. Replanning with rrts[C]//Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006. IEEE, 2006: 1243-1248.

[19] Otte M, Frazzoli E. RRTX: Asymptotically optimal single-query sampling-based motion planning with quick replanning[J]. The International Journal of Robotics Research, 2016, 35(7): 797-822.

[20] Jeong I B, Lee S J, Kim J H. Quick-RRT*: Triangular inequality-based implementation of RRT* with improved initial solution and convergence rate[J]. Expert Systems with Applications, 2019, 123: 82-90.

[21] Liao B, Wan F, Hua Y, et al. F-RRT*: An improved path planning algorithm with improved initial solution and convergence rate[J]. Expert Systems with Applications, 2021, 184: 115457.

[22] RRTX, [online] Available: https://github.com/potato-sauce/RRTX

[23] Prometheus autonomous UAV opensource project, [online] Available: https://github.com/amov-lab/Prometheus.