A Real-time Sampling-based Path Planning Algorithm for Unmanned Surface Vessels -- Torch

RO57015: Internship Report

Yuezhe Zhang

Delft University of Technology

TU Delft

A Real-time Sampling-based Path Planning Algorithm for Unmanned Surface Vessels -- Torch

by

Yuezhe Zhang

Student Name	Student Number
Yuezhe Zhang	5390664

University: TU Delft
Department: Cognitive Robotics
Supervisor: Javier Alonso-Mora
Email: j.alonsomora@tudelft.nl

Company: Demcon
Department: Demcon Unmanned Systems
Supervisor: Luuk van Litsenburg
Email: luuk.van.litsenburg@demcon.com

Project Duration: April 25, 2022 - September 30, 2022

TU Delft

Preface

This report is written to reflect the internship process corresponding to the RO57015 course of the Robotics master programme of the Delft University of Technology. A final presentation was given for the Demcon Unmanned Systems (DUS) team on 30/09/2022 and the group of Autonomous Multi-Robots Lab on 17/10/2022.

In this internship a lot of knowledge was gained on programming in c++ ,python, and the ROS framework. A deep understanding of sampling-based planner and implementation and visualization using ROS was gained as these were programmed from scratch in c++. Emphasis was laid on writing clean code.

A literature review and system architecture development were done to get a clear scope of the assignment and how the implementation should take place. Prototyping was done in python to show the first phase progress in the proposed sampling algorithm. Every month a meeting with other interns of DEMCON was held to discuss progress and give feedback on each others projects. In the final phase, implementation in cpp and simulation in ROS gave a good insight on the performance.

Special thanks are given to Luuk van Litsenburg for his guidance, insights and helping hand during the process, being the daily supervisor and Javier Alonso-Mora for being my supervisor at TU Delft. Gratitude goes to the the whole DUS team for this pleasant learning experience and good memories at the office.

Yuezhe Zhang
Delft, October 2022

Summary

Demcon Unmanned Systems manufactures and supplies unmanned vessels with autonomous capabilities. The vessels are equipped with sensors like LiDAR and GPS to build a mapping of the environment. Currently, a mission planner is used to specify the start and goal positions and the predefined path by the user, which requires the path planner to generate a safe and effective path avoiding the potential obstacles and following the predefined path. However, the current using Brute Force Branching and RRT algorithms in the path planner encounter issues in terms of computational efficiency and circumventing large obstacles. Therefore, The aim of this intern project is to design an efficient real-time sampling based path planning algorithm for the path tracking of unmanned vessels.

The proposed algorithm Torch can grow with specified parameters, including FoV, edge angle, edge length, and number of layers. The new structure allows the required samples to grow polynomially with the increase of number of layers, which addresses the exponential growth issue of Brute Force Branching. The algorithm can also be combined with RRT to reach further area, where the torch structure provides a warm start for the RRT and alleviates the stochastic inconsistent issue of RRT, which is beneficial when circumventing wide and long obstacles.

Tests were conducted in the CoppeliaSim Simimulator and ROS environment. The performance of three algorithms including Torch, Brute Force Branching and RRT is combined in static scenarios containing single obstacle, wide obstacle, long obstacle and multiple obstacles. Tests have shown that Torch algorithm can avoid single obstacles with fewer number of samples than the other two algorithms. Torch can be combined with RRT to circumvent 40m wide obstacle and 50m long obstacle. When the number of samples is less than 1200, the tree generating frequency is 10 to 50 Hz, when the number of samples is between 1800 and 2000, the frequency is 7 to 25 Hz. The computational efficiency and CPU consumption of Torch also outperforms the other two algorithms.

In the future, this work might be extended in the following directions. First, the current path planner should be combined with the dynamic obstacle tracking module to update the information about potential dynamic obstacles, which allows the vessels to navigate safely in a more dynamic environments. Second, the mass and drag of the robotic boat may change drastically when transporting goods. Therefore, incorporating an online model identification system and adaptive controllers will be one of our next steps. Third, wave disturbances always exist in natural waters or in fierce weather conditions, which will be addressed in the future controller design.

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

Abbreviations

Abbreviation	Definition
DUS	Demcon Unmanned Systems
BF	Brute Force Branching Tree
RRT	Rapidly-exploring Random Tree
ESDF	Euclidean Signed Distance Field

Symbols

Symbol	Description
V	List of vertices
E	List of edges
$G(V, E)$	Graph containing V and E
$\mathcal{Q}$	Configuration space
Q_F	Free configuration space
$\tau(\lambda)$	Continuous path that maps λ to a curve in $\mathcal{Q}$
Q_*	Goal configuration

1

Introduction

1.1. About Demcon Unmanned Systems

Demcon was established in 1993, and currently has more than 1000 employees in 16 companies and 11 locations in 4 countries. They work on solutions to social challenges in the areas of aerospace, agriculture &food, high-tech systems & materials, life sciences & health, smart industry and water & maritime. They do this by developing, manufacturing and supplying high quality technology and innovative products.

In 2017 Demcon Unmanned Systems (DUS), a subsidiary of Demcon, was established in Delft by a team with over 15 year of experience in developing, supplying and servicing industrial unmanned systems. Since then, they have developed and built high-tech electric, unmanned autonomous vessels for all kinds of waters and various maritime applications. The range of USVs that they offer can be seen in Figure 1.1. Demcon unmanned systems develops its own unmanned platforms, as well as autonomous navigation technologies to enable customers to conduct maritime operations and data acquisition at lower operating costs and in a safe, automated and sustainable manner. One of their biggest customers is Van Oord who uses DUS's vessels mainly for mapping the bottom of varies bodies of water as part of their 'VO:X' data program.

Demcon Unmanned Systems is divided up into three main groups: software, hardware and management. The software group consists of three people developing, testing and implementing software for the USV's. The software associated with DUS' products is a wide range of different applications and algorithms, including boat control algorithms, user interfaces and path planning algorithms. The hardware group focuses on the construction of USV's, from designing mechanical structures, ordering necessary equipments, performing system integration to repairing. Lastly, there is one manager responsible for acquisition, finance and marketing. He does so by meeting with potential customers, creating content for social media and keeping track of the finances. There are also several interns working on different projects.

Figure 1.1: The range of USVs that DUS offers [1]

1.2. Problem Description

Demcon Unmanned Systems builds unmanned vessels with autonomous capabilities. The vessels are equipped with sensors like LiDAR and Dual RTK-GPS, which are used to build a mapping of the environment. A mission planner is used to insert the start and goal positions by the user, which requires the path planner to generate a safe and effective path avoiding the potential obstacles.

This internship assignment is about designing an efficient real-time sampling based path planning algorithm for the path tracking of unmanned vessels. The internship providers want the path planner to fulfill the following properties, which guide the work in the rest of this report.

The first requirement for the path planning algorithm is the ability to follow the predefined path as close as possible and reach the destination. The predefined path is given by the user as a starting position and a set of continuous line segments through a user interface software. This is often the case when the user, such as Van Oord, wants the vessel to perform surveying on inland water and offshore inspection.

The second requirement for the path planning algorithm is the ability to avoid the obstacles when they get in the way of the predefined path. The collision avoidance is necessary since, in order to operate safely on the water, the automatic vessel has to avoid other vessels or obstacles and provides information on how to avoid them according to the rules of the water.

The third requirement for the path planning algorithm is the ability to perform in real-time. It means that the algorithm has to generate the path and updates in different timesteps, which makes it possible to avoid dynamic and unexpected obstacles. In addition, in order to be operated in real-time, the algorithm has to work efficiently and matches the control frequency.

The fourth requirement for the path planning algorithm is the ability to generate the path within certain angle constraints. The sampling-based path planning algorithm will generate samples randomly in the workspace, however, it usually ignores the angle constraint when it searches the path. Due to the fact that the current deployed controller on the vessel can not deal with very aggressive turning angles, and it is suggested to keep the angle within 15 degrees.

2 Methodology

2.1. Related Work

General framework for motion planning
The motion planning problem for a robot with d degrees of freedom can be described as finding a trajectory for a point indicating the robot's configuration through a d-dimensional configuration space. It can be more formally specified by a configuration space $\mathcal{Q} \subset \mathbb{R}^d$, a constraint $F : \mathcal{Q} \to \{0, 1\}$, an initial configuration $q_0 \in \mathcal{Q}$ and a set of goal configurations $Q_* \subset \mathcal{Q}$. The feasible configuration space is a subset of the configuration space $\mathcal{Q}$ that satisfies the constraint $Q_F = \{q \in \mathcal{Q} | F(q) = 1\}$. The objective is to find a continuous path $\tau : [0, 1] \to \mathcal{Q}$ such that $\tau(0) = q_0, \tau(1) \in Q_*$, and $\forall \lambda \in [0, 1], \tau(\lambda) \in Q_F$. The simplest motion planning problems require the robot to move through space without colliding, where the free-space motion is constraint by $F(q)$.

Motion planning is PSPACE-hard as shown in [2], which suggests in the worst case, exponential time is required for any algorithm to solve the problem. However, there are exact algorithms that leverage algebraic geometry to solve problems using only polynomial space to prove it is PSPACE-complete [3]. In addition, there are two most widely approaches to solve the motion planning problem: sampling-based method ([4], [5]) and optimization-based method ([6], [7]). Both classes of algorithms are useful in practice, but are not complete since they can not identify infeasible problems.

Why choose sampling-based method
One of the advantages of sampling-based algorithms is that some algorithms can be shown to be probabilistically complete under certain robustness conditions, meaning that if there exists a solution, the probability that they will fail to find a solution will be zero as the running time increases. This is the main reason that the DUS company chooses to implement the sampling-based method for the path-tracking problem. In addition, sampling-based method is much easier to implement and requires less computing resource when collision checking, where collision checking module can provide information about feasibility of candidate trajectories instead of constructing an explicit representation of the obstacle-free environments.

Why choose single-query method
Two subcategories can be distinguished among the sampling-based methods: single-query and multi-query algorithms. A single-query algorithm, such as Rapidly-Exploring Random Tree (RRT)[5], creates a graph specifically for the start and goal configuration, if a new start and goal configuration are updated, a new graph needs to be created. A multi-query algorithm, such as Probabilistic Roadmap (PRM)[4], constructs a graph that can be used for multiple queries. Even though multiple-query methods are valuable in highly structured environments, such as factory floors, most online planning problems do not require multiple queries, since, for instance, the robot moves from one environment to another, or the environment is not known a priori. Moreover, in some applications, computing a roadmap a priori may be computationally challenging or even infeasible.

Previous work about offline RRT
One of the relevant work about sampling-based motion planning is the group project I did with my teammates in the course *RO47005 Planning and Decision Making* as shown in Figure 2.1 [8]. In the project, a hierarchical framework combing RRT* and non-linear MPC was constructed for path planning of a drone, where the RRT* works as a global planner to find a global path from a starting point to a goal, and *minimum snap optimization* was used to smooth the trajectory and perform smart time allocation, and finally the non-linear MPC works as a local planner to follow the pre-computed global path and avoid unexpected obstacles online. The global planning of RRT* was carried out in $\mathbb{R}^3$, thanks to the property of differential flatness that reduces from the configuration space $\mathbb{R}^3 \times SO(3)$ to $\mathbb{R}^3 \times \mathcal{S}$ and the further assumption that ignores the yaw angle.

However, the global planning of RRT* was run offline and the global path was generated previously to the execution which requires the local planner to enable the reactivity. The hierarchical framework increases the overall complexity of the system. It will be shown later that the RRT can also be performed online to provide a more efficient solution while keeping the local reactivity.

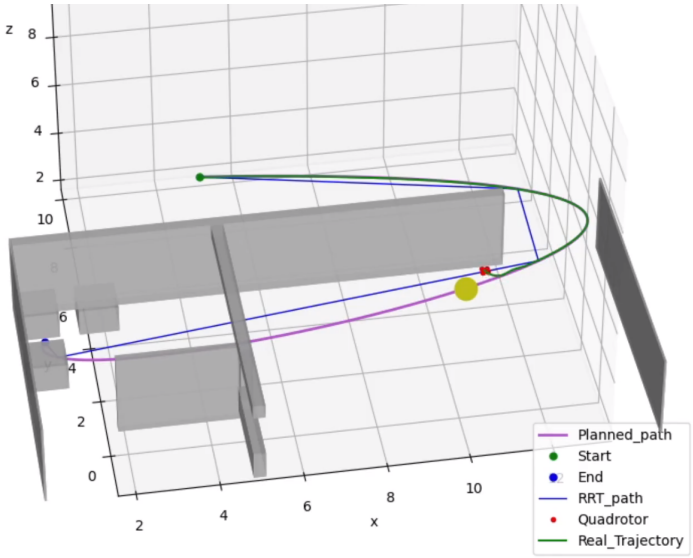
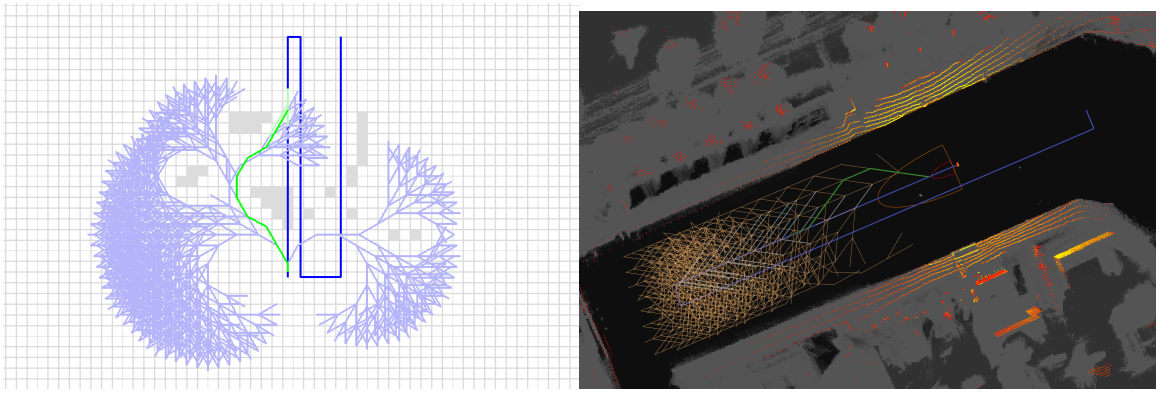


Figure 2.1: Previous work in drone

Previous work about brute force branching tree
Previous work has been done at DUS about brute force branching tree. An outline of the algorithm is given in Algorithm 1. The algorithm is initialized with a graph that includes the initial state as its single vertex, and no edges. At each iteration, a vertex x_0 is popped out from V, and the function GenerateVertices will generating vertices starting from x_0, ranging in a certain degrees, such as -30 degrees to 30 degrees. Then the collision checker will check if these edges are collision free, if so, they will be added to the graph. The GenerateConnection part will help the tree to steer back to the referenced path, which will be explained in details in the next section.
The biggest issue of brute force branching tree is that the number of vertices needed will grow exponentially as the tree layer grows, as can be shown in Figure 2.2, which is inefficient when encountering big obstacles and thus making it hard to find solutions.

Algorithm 1 Brute Force Branching

Given referenced path $\tau*$;
$V \leftarrow \{x_{init}\}, E \leftarrow \emptyset$;
for $i = 1$ to N **do**
  $x_0 \leftarrow$ PopOutVertex(V);
  $x_1, x_2, x_3 \leftarrow$ GenerateVertices(x_0);
  for $j = 1$ to 3 **do**
    if ObstacleFree(x_0, x_j) **then**
      $V \leftarrow V \cup \{x_j\}$;
      $E \leftarrow E \cup \{(x_0, x_j)\}$;
    end if
  end for
  if GenerateConnection($G = (V, E), \tau*$) **then**
    $V \leftarrow V \cup \{x_{on\_line}\}$;
    $E \leftarrow E \cup \{(x_{connection}, x_{on\_line})\}$;
  end if
end for
Return $G = (V, E)$



(a) Prototype in python **(b)** Rviz in ros and cpp

Figure 2.2: Previous work about brute force branching tree

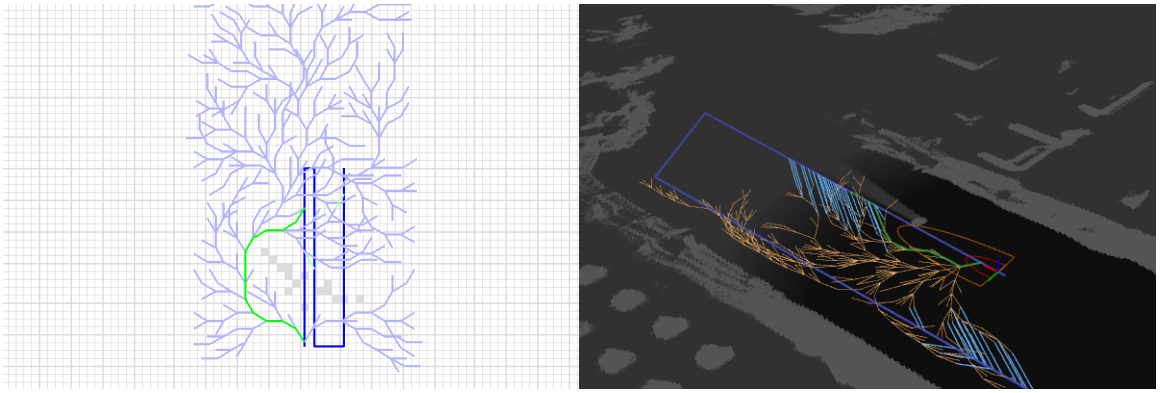
Previous work about RRT

Previous work has been done at DUS about RRT. An outline of the algorithm is given in Algorithm 2. The algorithm is initialized with a graph that includes the initial state as its single vertex, and no edges. At each iteration, a vertex x_{rand} is sampled in a window in front of the vessel, and an attempt is made to find the nearest vertex $x_{nearest}$ to x_{rand}. Then the steering function will try to extend edge from $x_{nearest}$ to x_{rand} and the collision checker will check if the edge between $x_{nearest}$ and x_{rand} is collision free, if so, it will be added to the tree. The GenerateConnection part will help the tree to steer back to the referenced path, which will be explained in details in the next section.

RRT still encounters some issues in the real-time path tracking due to the nature of stochastic property, it is difficult to generate consistent solutions at different timesteps in real-time. And the sampling window will also be blocked to some extent when encountering big obstacles, making it difficult to plan in these extreme cases. And it is also not efficient to sample in the whole space of the window if there are only small obstacles in the way.

Algorithm 2 RRT

Given referenced path $\tau*$;
$V \leftarrow \{x_{init}\}, E \leftarrow \emptyset$;
for $i = 1$ to N **do**
  $x_{rand} \leftarrow$ SampleFreeInWindow;
  $x_{nearest} \leftarrow$ Nearest($G = (V, E), x_{rand}$);
  $x_{new} \leftarrow$ Steer($x_{nearest}, x_{rand}$);
  if ObstacleFree($x_{nearest}, x_{new}$) **then**
    $V \leftarrow V \cup \{x_{new}\}$;
    $E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$;
  end if
  if GenerateConnection($G = (V, E), \tau*$) **then**
    $V \leftarrow V \cup \{x_{on\_line}\}$;
    $E \leftarrow E \cup \{(x_{connection}, x_{on\_line})\}$;
  end if
end for
Return $G = (V, E)$



(a) Prototype in python **(b)** Rviz in ros and cpp

Figure 2.3: Previous work about RRT

2.2. Main Framework for Path Tracking

The path-tracking problem is described as follows. The input is an occupancy grid map, an initial state of the vessel (including the position and orientation), a goal state of the robot and a referenced path, which is specified by the user via a user interface software. The desired output of the planner is a trajectory (sequence of states) that is safe, near-minimal in length and with less skipped distance from the predefined path. It is worth mentioning that the planning module and the control module is separated, which means the smoothing of the path or the kinematic constraints are achieved and satisfied by the control module and thus are not considered in this report.

The main steps to solve the path tracking problem can be framed as 5 steps as shown in Figure 2.4. The first steps is to convert the occupancy grid to Euclidean Signed Distance Field (ESDF). Since the occupancy grid is not directly usable and suitable for continuous motion planning, and converting it into an ESDF enables each pixel to indicate the distance to the closest obstacle, which is especially beneficial when collision checking. After getting the information about the obstacle distance, the second step is to check if the initial plan is collision-free. The collision-free is determined by the ESDF map and obstacle threshold distance, if the obstacle lies far from the threshold, then it is collision-free. If the referenced path is collision-free along the way, the path planner will just return the referenced path and then the controller will just follow it until the vessel reaches the destination. The third and fourth step will be involved if there is obstacle in the way or close to the path. This will only be triggered if there

is obstacle within a certain distance, called *forward checking distance*, in front of the vessel. Then in order to evade the obstacle, an initial waypoint will be created and the tree will be generated starting from the point. The process of generating tree will stop until the vessel steers back to the referenced path. Finally when the vessel reaches the destination, the final mission is completed the vessel will stop.

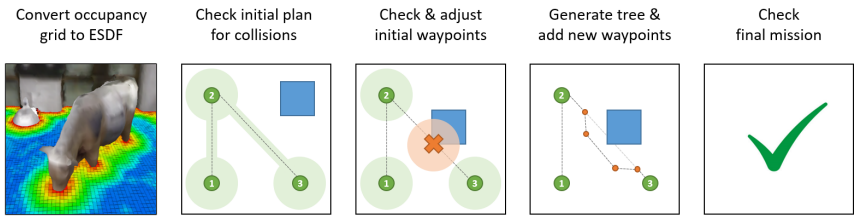


Figure 2.4: Main Steps for Path Tracking Planner

It is worth mentioning that this framework can apply to different tree generating algorithm, such as the aforementioned brute force branching tree and RRT. The newly developed tree generating algorithm Torch will also be used within this framework.

2.3. Torch Algorithm

A new sampling-based tree generating algorithm is proposed to address the issues of brute force branching tree and RRT. It is called Torch, because the generated tree look similar to the torch that burns fire on top and gives light.

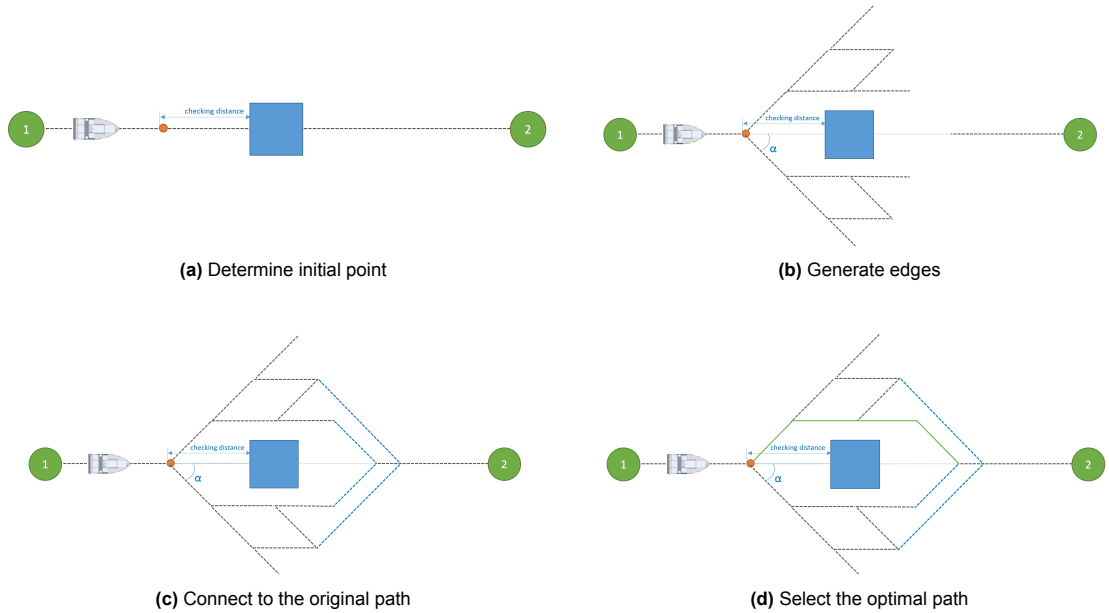
An outline of the algorithm is given in Algorithm 3. The algorithm is initialized with a graph that includes the initial state as its single vertex, and no edges. The tree will be initialized with the first layer of tree ranging from certain angles, determined by *initialized angle* θ. If θ is set to 60, it means the tree can range from -60 degrees to 60 degrees, which also means the *Filed of View(FoV)* is 120 degrees. If the angle between the edges α is set to 15 degrees, then the first layer will generate 9 possible edges. θ should be chosen that can be divided by α. After the first layer is initiated, the algorithm will enter into a loop as long as the tree layer is less than *max torch layer*. At each iteration, a vertex x_0 is popped out from V, and the function GenerateTorchVertices will generating vertices starting from x_0, ranging in a certain degrees, such as -30 degrees to 30 degrees.The functionality of CollisionChecking and GenerateConnection remain the same with the previous algorithm.

The biggest difference between the proposed Torch algorithm with the brute force branching algorithm lies in that they generate vertices in a different way. The functionality GenerateTorchVertices ensures the required samples will increase as a polynomial of the number of generated layer, instead of exponential one. This greatly increases the speed and efficiency of the planning algorithm. In addition, the Torch algorithm can be combined with RRT algorithm to reach further distance via the *stopping waypoints* V_{stop}. The *stopping waypoints* will be stored once the layer is larger than specified *minimum specified rrt layer*, then the RRT algorithm will generate trees starting from these stopping waypoints. Thus, Torch provides a good initial warmstart for the RRT, this is is especially helpful when the vessel wants to surpass a wide obstacle and can alleviate the issue of stochastic consistency.

The animations of the tree generating process of Torch algorithm can be shown in Figure 2.5 and Figure 2.6. In the animation, the initial angle θ and α are chosen the same for simplicity. Figure 2.5 shows the process in timestep 1, which involves determining the initial point, generating edges, connecting to the original path and selecting the optimal path.

Algorithm 3 Torch Algorithm

Given referenced path $\tau*$;
$V \leftarrow \{x_{init}\}, E \leftarrow \emptyset, L \leftarrow 0, V_{stop} \leftarrow \emptyset, \theta \leftarrow 60°, \alpha \leftarrow 15°$;
InitializeTree(x_{init}, θ, α);
while $L < max\_torch\_layer$ **do**
  $x_0 \leftarrow$ PopOutVertex(V);
  $L \leftarrow$ GetDepth(x_0);
  if $L > min\_rrt\_layer$ **then**
    $V_{stop} \leftarrow V_{stop} \cup \{x_0\}$
  end if
  $x_1, x_2 \leftarrow$ GenerateTorchVertices(x_0, α);
  for $j = 1$ to 2 **do**
    if ObstacleFree(x_0, x_j) **then**
      $V \leftarrow V \cup \{x_j\}$;
      $E \leftarrow E \cup \{(x_0, x_j)\}$;
    end if
  end for
  if GenerateConnection($G = (V, E), \tau*$) **then**
    $V \leftarrow V \cup \{x_{on\_line}\}$;
    $E \leftarrow E \cup \{(x_{connection}, x_{on\_line})\}$;
  end if
end while
Return $G = (V, E)$



(a) Determine initial point

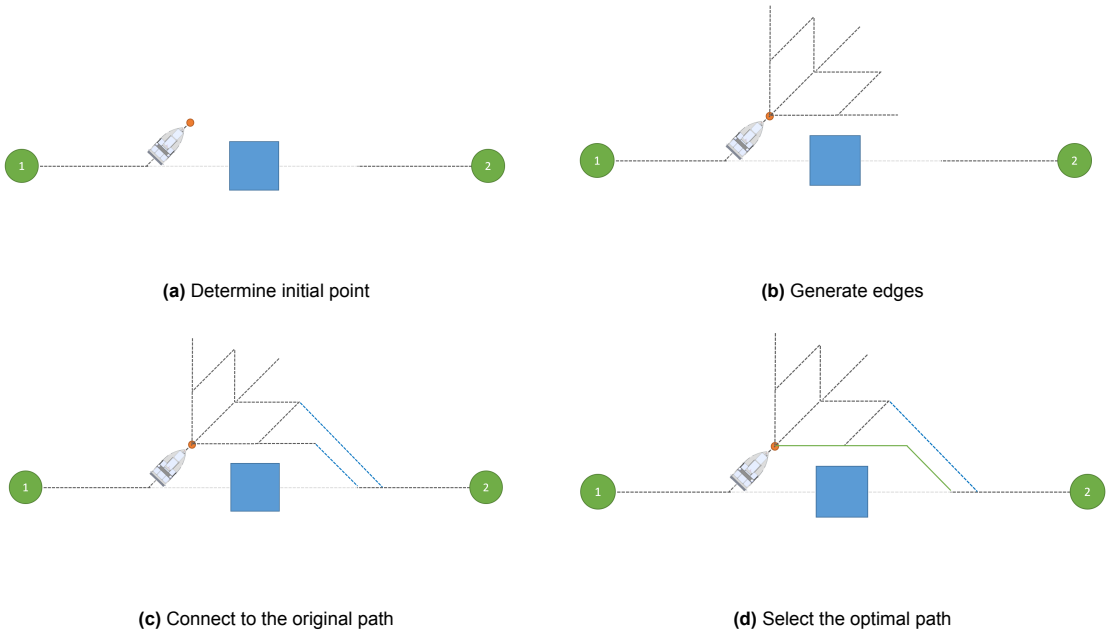
(b) Generate edges

(c) Connect to the original path

(d) Select the optimal path

Figure 2.5: Torch in timestep 1

At the next timestep as shown in Figure 2.6, when the vessel travels on one of the edges and deviates from the referenced path, the initial point will be updated to the nearest intersection point, and the tree will be generated at a different direction. The following steps of connecting to the original path and selecting the optimal path remains the same.



(a) Determine initial point

(b) Generate edges

(c) Connect to the original path

(d) Select the optimal path

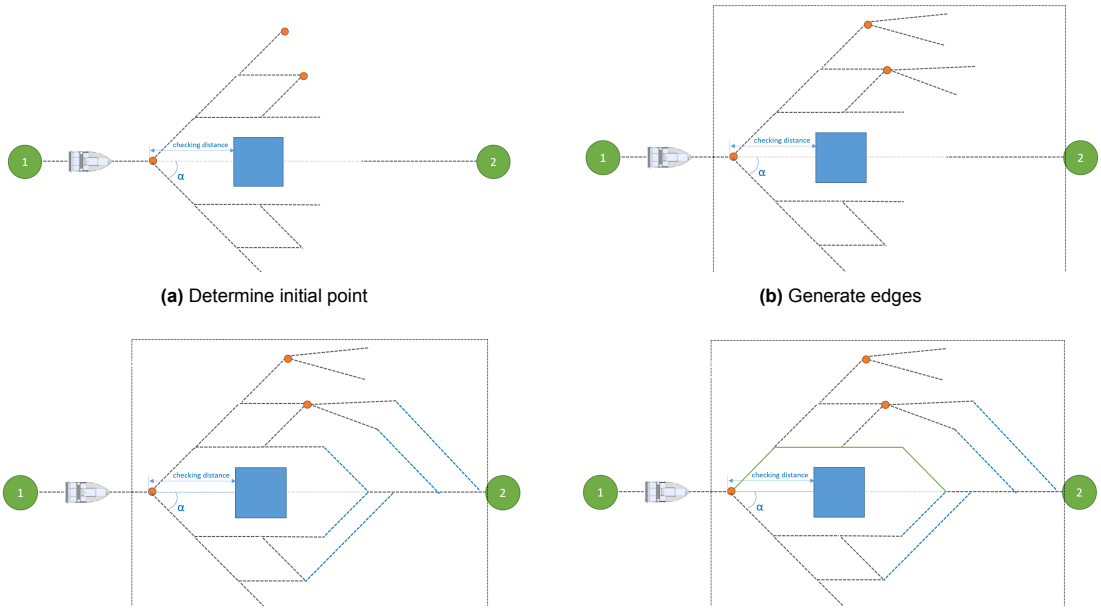
Figure 2.6: Torch in timestep 2

When combing the Torch algorithm and RRT, an outline of the algorithm is given in Algorithm 4. After initializing the tree, the algorithm is in a loop. At every iteration, the algorithm will check if the current vertex's number of layer is less than the *max torch layer*, if so it will generate Torch tree, otherwise it will generate RRT tree.

Algorithm 4 Torch-RRT

Given referenced path $\tau*$;
$V \leftarrow \{x_{init}\}, E \leftarrow \emptyset, L \leftarrow 0, V_{stop} \leftarrow \emptyset, \theta \leftarrow 60°, \alpha \leftarrow 15°$;
InitializeTree(x_{init}, θ, α);
for $i = 1$ to N **do**
  if $L < max\_torch\_layer$ **then**
    GenerateTorchTree;
  else
    GenerateRRTTree;
  end if
end for
Return $G = (V, E)$

The animation of the Torch-RRT is shown in Figure 2.7. The difference between this one and the Torch algorithm lies in the second phase, where the RRT tree is generated. The samples of RRT are randomly taken from a moving window in front of the vessel. The window is determined by the parameters *width*, *height* and *offset*, where the *offset* determines how far the window is from the vessel, which can also be negative.



(a) Determine initial point

(b) Generate edges

(c) Connect to the original path

(d) Select the optimal path

Figure 2.7: Torch RRT

3

Experiments and Results

3.1. Software structure

This section will discuss the general software architecture of the path planner. The architecture is based on ROS and CPP. The path planner consists of 5 main classes, whereof 3 are modular. These classes will execute the most important tasks of the path planner and have a lot of intercommunication between them. The diagram (Figure 3.1) below shows the general architecture in a schematic way. The non-modular classes are Planning Manager and Path Generator, the modular classes are Tree, Collision Detector and Solution Chooser, which can be replaced by other components.

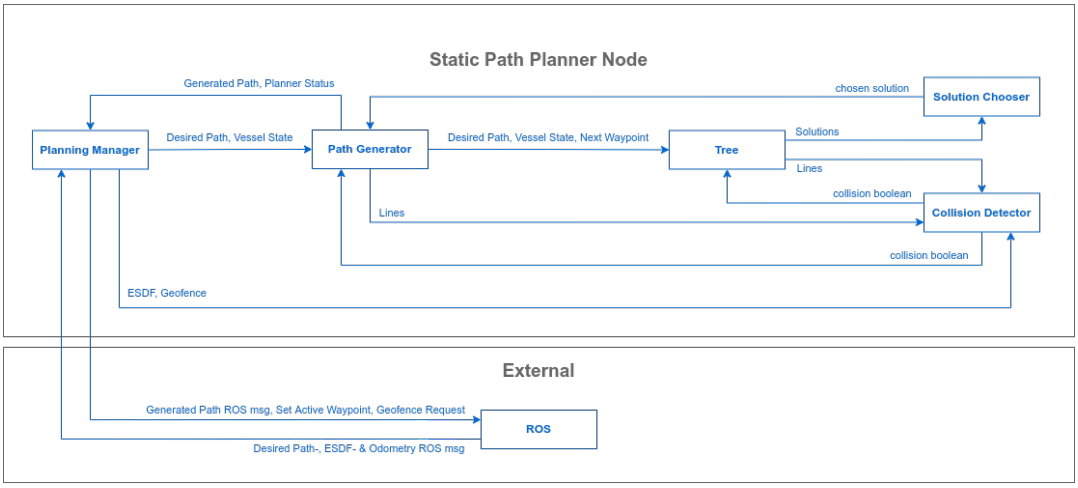


Figure 3.1: Software Structure

Each ROS node will have the same Planning Manager and Path Generator, but can choose a specific type of the Tree, Collision Detector and Solution chooser classes. An example would be the static brute branching node, which is comprised of a static collision detector, a static solution chooser and a brute branching tree. The next part will explain each of the 5 classes in more details starting with the Planning Manager Class.

3.1.1. Non-modular Classes

Planning Manager
The planning manager has the responsibility of overseeing the whole Path Planning process. The most important subtask of serving as the communication layer between ROS and the rest of the path planner. It will create the subscribers and publishers and acquire and transmit all ROS msgs. The ROS msgs

are communicated with the other classes. In order to do this it will convert the ROS msgs to a format suitable for usage by the other Classes.

The planning manager will obtain the ESDF grid, Odometry, Desired Path and Geofence msgs from other ROS packages. The ESDF grid msg is converted to an ESDF grid object and passed on to the Collision Detector. The Odometry msg is converted to a Vessel State object, which is passed on to the Path Generator. The Desired Path msg is converted to a Path object and passed on the Path Generator as well. The planning manager will also request the flight computer to send a Geofence msg, this msg in turn is received and converted to a Geofence object. The Geofence object is passed on to the Collision Detector.

The planning manager will also obtain the ROS parameters and load them into a Parameters object. This Parameters object is accessible for all main 5 Classes. The Planning Manager will also keep track of the global planner Status.

Furthermore, it will also contain the main callback function for the path planner. In this callback function it will update the ROS params and geofence and planner status. It will also update the active mission waypoint if needed, but most importantly it will ask the Path Generator to check if an evasion is required. If it is required it will receive a Path that does not contain NAN values from the Path Generator. This received Path object then contains an evasion around the potentialy detected collision. This path is in turn published back to the ROS system.

The path generator will communicate to the Planning Manager is the active waypoint should be set, by changing the connection mission index in the Path object to a value above 0. The value of the connection mission index represents the waypoint number to be set as active.

Path Generator
The Path Generator will check if an evasive path is required by checking the desired path onward from the current location to a certain distance forward using the Collision Detector. If an evasion is required it will instruct the Tree to generate many evasive path options, which we will call solutions. It will then pass these solutions to the Solution Chooser, which as the name implies, will choose a solution from the many solutions. The Path Generator will also keep the previous solution in its memory and pass it on to the Solution Chooser as well. By doing this we will not change solution all the time, which is especially important for the more stochastic type of Trees. It will also update this kept solution and check it again for collision at the new time step using the Collision Detector. The Path Generator will also determine the start point for the tree and pass it to the Tree together with the ESDF and Vessel State.

3.1.2. Modular Classes

Tree
The Tree class is the first of the modular classes to be mentioned. It is responsible for finding evasive paths around obstacles and returning the solutions. The generation of the tree is done by extending it through the generation of samples. The start point for the tree generation is provided by the Path Generator.

The waypoints that are added to the tree due to the samples are called vertices. The connections between vertices are called edges or branches. Each edge is checked for potential collisions using the Collision Detector. The new vertice corresponding to the edge is only added if no collision is detected.

The path planning is of the Sampling Based type due to the fact that we create a tree by sampling in a certain way. The way of sampling can vary greatly and therefore this class has been made modular. There is a base Tree class from which all Tree subclasses can inherit. Each node can choose a tree type, which will greatly affect the path planner behavior, including brute force branching tree, RRT and Torch.

Collision Detector
The Collision Detector class is also one of the modular classes. It is used to determine whether following specific path segments will result in collisions. It can therefore be used to generate a collision free tree and check other paths (e.g. previous solution) for collisions as well. This class is made modular as we might want to add different detectors later on, which could handle different environments. For example one detector would be able to only handle static environments, while the other could also handle dynamic environments. At the moment the only subclass of the Collision Detectors is the Static Detector. All detectors receive an array of Line objects to evaluate and return whether there is a potential collision by means of a single boolean.

The Static Detector checks the vector of Line objects for potential collision using two sources: ESDF and geofences. The lines are extended for the ESDF by a certain distance in order to prevent the emergency brake from activating even though the unextended line does not contain an obstacle. This extended line is then converted to an array of corresponding grid cells for the ESDF using a line drawing algorithm. Each cell of the array is then checked for in the ESDF grid for its value. The value should be larger for each cell then the set minimum ESDF threshold distance value. If the values are all larger it means that no obstacle should be closer than the set minimum ESDF threshold distance value. Therefore, according to the ESDF there will be no obstacles. If one value is smaller than the threshold it means that there will be an obstacle.

The geofence is indirectly obtained from the flight computer and converted to a geofence object. In this conversion the geofence is also offset inward or outward, which depends on whether it is an inner or outer geofence. The amount of offset is determined by a ROS param. The extension distance that was used for the ESDF is NOT applied here as the emergency brake does not see geofences. The offset ROS parameter already takes into account some safety margin. The lines to be checked are discretized into points and it is checked for each point if it violates the geofence. If no geofence violations occur then no potential collision is returned.

The returns from the ESDF and Geofence checks are combined in order to return the total result for a potential collision indication.

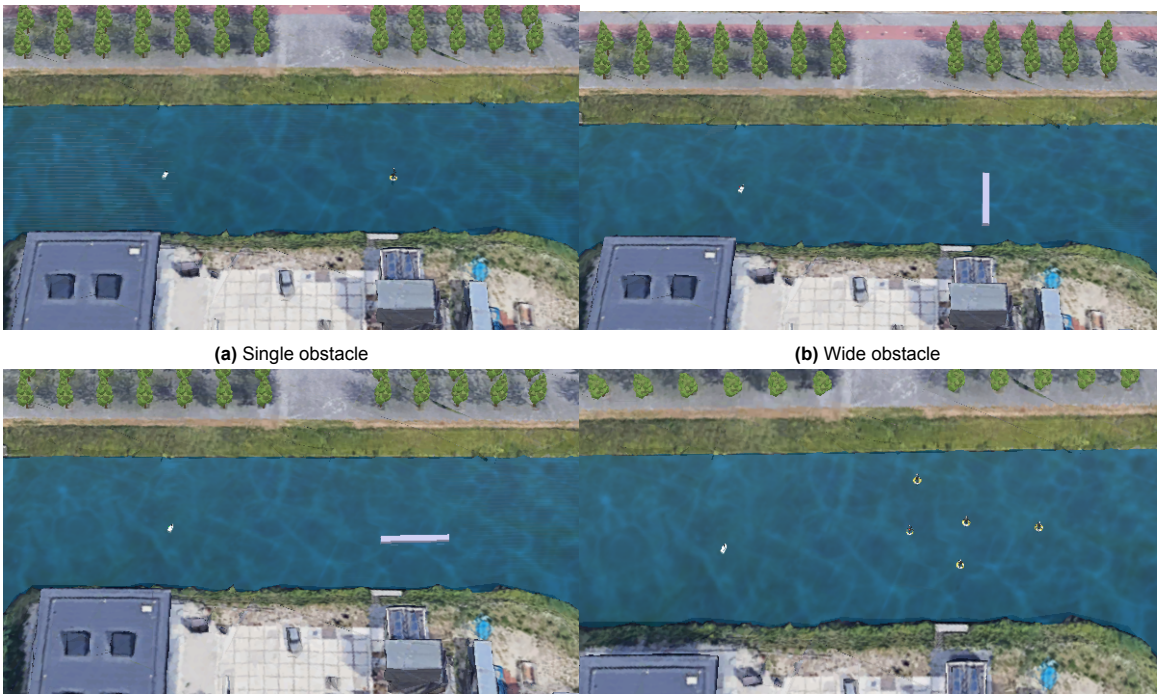
Solution Chooser
This class will obtain the many potential evasive paths and choose one from them. In order to do this it will calculate several scores for each path and sum them. The path with the lowest score is chosen. This class is also a modular one and each subclass will at least have to provide a function to calculate the total score for an evasive action (solution). Each subclass can then define scores and sum them internally. At the moment only the Static Chooser subclass exists.

The static chooser calculates three scores for each potential best solution. The scores are then summed and the best solution is chosen. The first score is the path length score. The path length score calculates the length of the path and uses it to calculate the score. The score and the path length are linearly correlated with no bias. A length which will be two times longer will therefore result in a score that is twice as high. A longer path is therefore penalized. The second score is the distance skipped score. The amount of distance of the original desired path that is skipped due to the evasion is calculated. The score and the distance skipped are linearly correlated with no bias. A distance skipped which will be two times longer will therefore result in a score that is twice as high. The calculation is done by looking at the desired lines segments that still remain and assuming that it is completely skipped. Then the distance is reduced by the segments that are covered by the evasion and the segments after the evasion. A path with more distance skipped is therefore penalized. The third score is the left-right score, which should allow us to prefer a clockwise evasion over a counter-clockwise one or vice-versa. Currently, this score is however not working properly and should be disabled.

3.2. Simulation Scenarios

For simulation testing, the CoppeliaSim robotics simulator was used. CoppeliaSim is used for fast algorithm development, factory automation simulations, fast prototyping and verification, robotics related education and much more [9]. Each model can be individually controlled via an embedded script, a plugin or a ROS node, which makes CoppeliaSim very versatile and ideal for multi-robot applications. Controllers can be written in C/C++, Python, Java, Lua, Matlab or Octave.

A 3D environment imitation model of the Green Village at the Delft University of Technology campus was created in Coppeliasim at DUS. In this replica, it is possible to place vessels, static or dynamic obstacles in the form of different shaped and sized models. As can be seen in Figure 3.2, several simulation scenarios are created considering different type of static obstacles. Only the static obstacles are considered here since the company wants another intern to implement the detection and tracking of dynamics obstacles using Kalman filtering and it is still ongoing and thus the testing performance will be combined with his work and mine in the future. However, it will be shown in the next section, the planning framework DUS uses now and the planning algorithm I propose can be easily extended to the case where dynamic obstacles are involved, which only requires the updates of the obstacle information after detection and tracking the dynamic obstacles.



(a) Single obstacle **(b)** Wide obstacle

(c) Long obstacle **(d)** Multiple obstacles

Figure 3.2: Different simulation scenarios

Figure 3.2 shows four types of simulation scenarios considered in this report, including single obstacle, wide obstacle, long obstacle and multiple obstacles. The obstacles in scenario - single obstacle and scenario - multiple obstacles are represented by static buoys. The obstacles in scenario - wide obstacle and scenario - long obstacle are represented by rectangles with different length positioned vertically or horizontally on the referenced path. The scenarios with long obstacle and wide obstacle are used to test the how wide and far the vessel is capable to circumvent large obstacles, which can be easily encountered in the real environments such as big ships or shores.

3.3. Simulation Results

3.3.1. Scenario 1

In the first scenario, the algorithms are tested in the simulated Green Village with a single buoy obstacle. The performance of three algorithms including Torch, Bruteforce Branching and RRT is compared.

Some fixed common parameters are shown in Table 3.1. Among these parameters, the *initialized angle* θ for Torch is set to $60°$, and thus the FoV is $120°$. The checking distance is set to 10, since the single obstacle is easy to detect and make in-time action in a short detection distance. The minimum and maximum lengths of connection are set to 3 and 10, which means the functionality GenerateConnection will create connection edges from the tree to the referenced path with the length ranging from 3 to 10. And the angle between the connection edge and the referenced path is set to $30°$

Parameters	Values
FoV of Torch	$120°$
checking distance	10m
min length of connection	3m
max length of connection	10m
connection angle	$30°$

Table 3.1: Fixed parameters for scenario 1

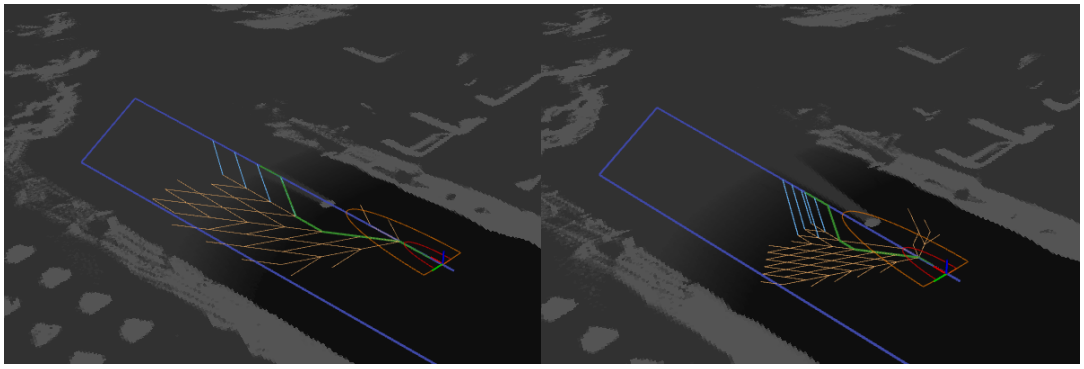
The simulation experiments in scenario 1 mainly compares the performance of the three algorithms with different angle α, edge length L, number of samples N. Two metrics are used to evaluate the performance of different algorithms, including the time of tree generation and the CPU consumption of the software. The parameters and results are shown in Table 3.2.

Scenario	Tree type	α (Angle)	L(Edge)	N(samples)	T(Tree Generation)	CPU
1-1	Torch	30	4	100	<20ms	5.5-8.5
1-2	Torch	30	2	300	<20ms	14-17
1-3	Torch	15	4	450	20ms	10-15
1-4	Torch	15	2	800	20-40ms	10-15
1-5	BF	30	4	100	<20ms	5.5-8.0
1-6	BF	30	2	300	<20ms	5-10
1-7	BF	15	4	450	20-40ms	10-15
1-8	BF	15	2	800	20-100ms	10-16
1-9	RRT	30	4	850	80-150ms	15-18
1-10	RRT	30	2	800	60-100ms	30-45
1-11	RRT	15	4	800	80-140ms	50-70
1-12	RRT	15	2	1000	60-160ms	50-70

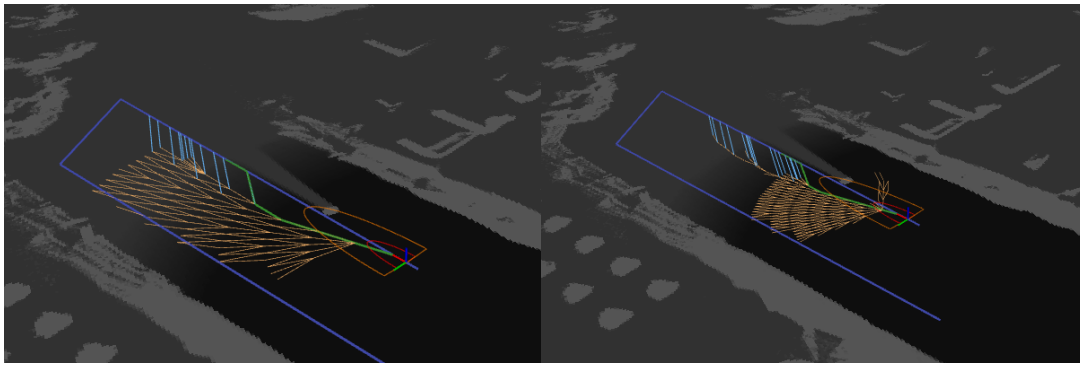
Table 3.2: Results in scenario 1

The screenshots of performance in RViz of the three algorithms in single obstacle scenario are shown in Figure 3.3, Figure 3.4 and Figure 3.5. In Figure 3.3, the Torch algorithm requires more samples if the edge length and the angle α are smaller. This is because more samples are required to cover the same exploration space when the length and angle are small. Torch also achieves a computational frequency at around 25 - 50 Hz, which outperforms the other two algorithms.

Compared with Brute Force Branching, Torch covers large area and achieves less overlap at the same number of samples. In addition, it achieves better computational performance regarding to the tree generation time and the CPU consumption.

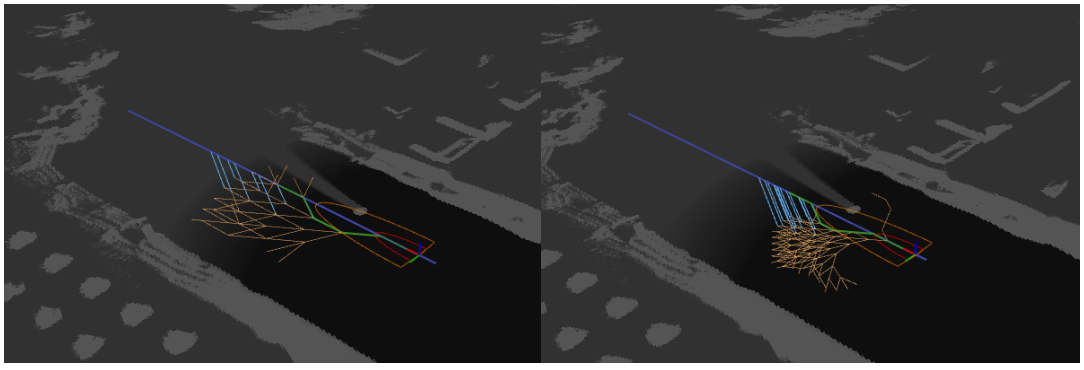


(a) 1-1 **(b)** 1-2

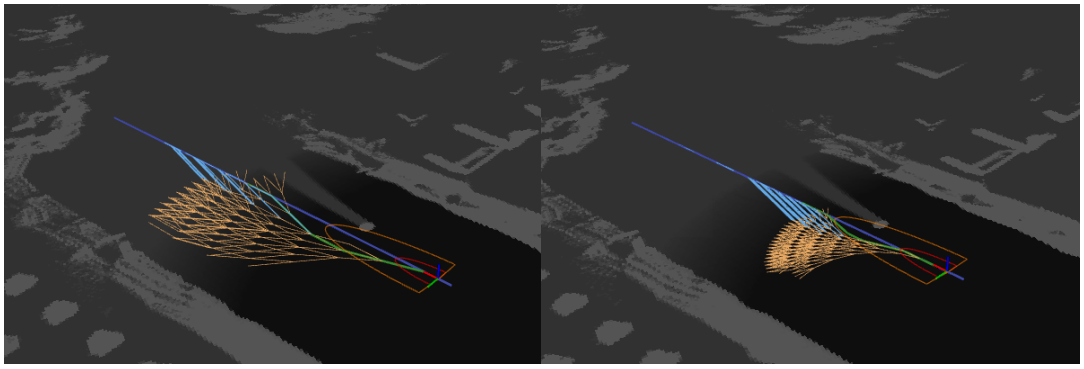


(c) 1-3 **(d)** 1-4

Figure 3.3: Performance of Torch for scenario 1



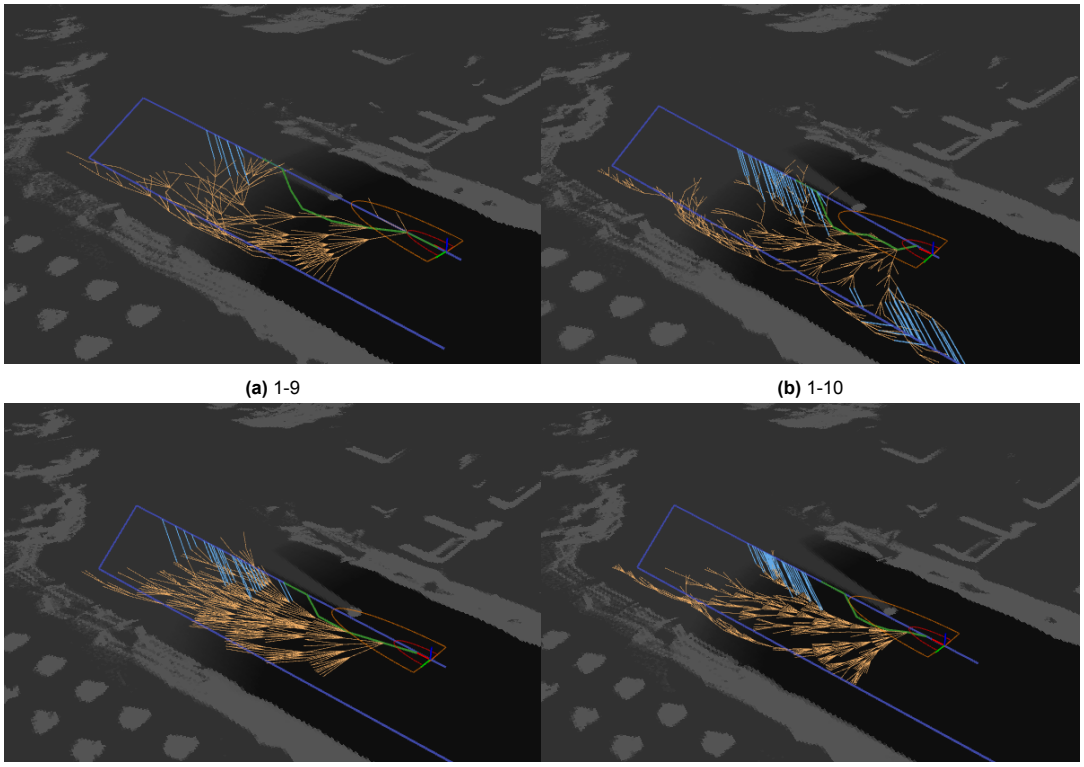
(a) 1-5 **(b)** 1-6



(c) 1-7 **(d)** 1-8

Figure 3.4: Performance of Brute Force for scenario 1

Compared with RRT, Torch requires less samples to find a solution and it is also more computationally efficient. In addition, it does not need to sample in a very large area, which addresses the issue that much effort is done in the non-relative area.



(a) 1-9 **(b)** 1-10

(c) 1-11 **(d)** 1-12

Figure 3.5: Performance of RRT for scenario 1

3.3.2. Scenario 2

In the second scenario, the algorithms are tested in the simulated Green Village with varying width of wide obstacles. The performance of three algorithms including Torch-RRT, Bruteforce Branching and RRT is compared.

Some fixed common parameters are shown in Table 3.3. Among these parameters, the *initialized angle* θ for Torch is set to $15°$, and thus the FoV is $30°$. The minimum and maximum lengths of connection are set to 3 and 10, which means the functionality GenerateConnection will create connection edges from the tree to the referenced path with the length ranging from 3 to 10. And the angle between the connection edge and the referenced path is set to $30°$

Parameters	Values
FoV of Torch	$30°$
min length of connection	3m
max length of connection	10m
connection angle	$30°$

Table 3.3: Fixed parameters for scenario 2

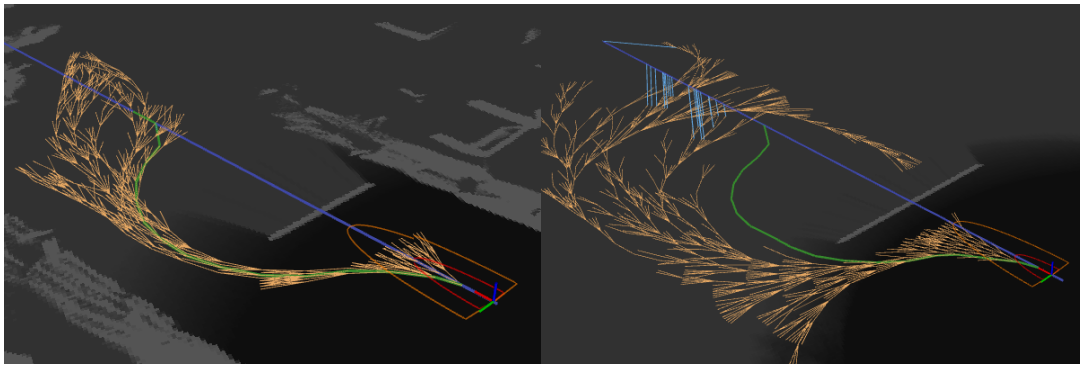
The simulation experiments in scenario 2 mainly tests the ability of different algorithms to circumvent wide obstacles. Thus, the performance of the three algorithms with different angle α, edge length L, number of samples N and checking distance d is tested in front of different width obstacles. Two metrics are used to evaluate the performance of different algorithms, including the time of tree generation and the CPU consumption of the software. The parameters and results are shown in Table 3.4.

Scenario	Tree type	w(Width)	α (Angle)	L(Edge)	d	N(samples)	T	CPU
2-1	Torch-RRT 3, 1	10	15	2, 3	15	1200	20-80ms	30-70
2-2	Torch-RRT 6, 4	20	15	2, 3	18	1500	40-100ms	20-50
2-3	Torch-RRT 6, 4	30	15	2, 3	25	2500	80-180ms	50-90
2-4	Torch-RRT 6, 4	40	15	2, 3	35	3500	140-340ms	70-100
2-5	BF	10	15	3	20	3000	60-220ms	40-95
2-6	BF	20	15	4	20	10000	120-320ms	40-100
2-7	BF	30	15	5	20	50000	1000-1400ms	95-100
2-8	RRT	15	15	2,3	20	1500	40-160ms	30-50
2-9	RRT	20	15	2,3	20	2000	40-180ms	30-70
2-10	RRT	30	15	2,3	20	3000	80-360ms	50-100

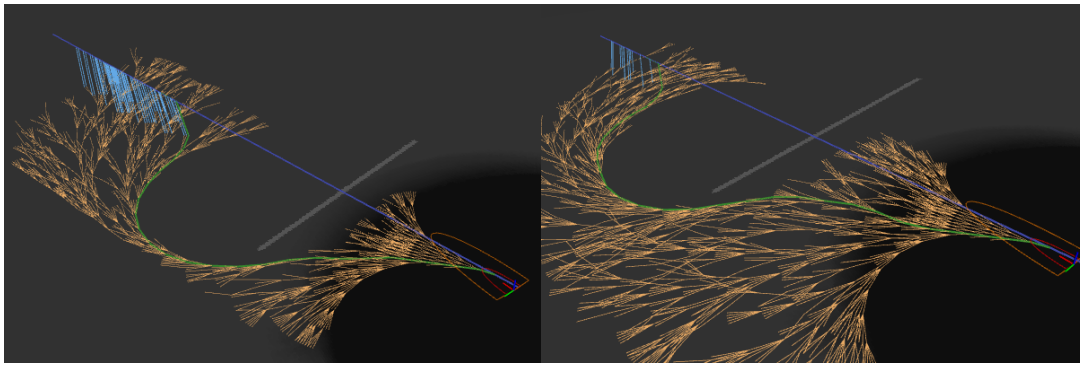
Table 3.4: Results in scenario 2

The screenshots of performance in RViz of the three algorithms in wide obstacle scenario are shown in Figure 3.6, Figure 3.7 and Figure 3.8. In Figure 3.6, the Torch-RRT algorithm is shown to successfully circumvent the wide obstacle up to 40 meters wide. To circumvent a wider obstacle, it requires a longer checking distance d, because the algorithm should have enough time and distance to make actions when it perceives an obstacle. In Torch-RRT, Torch provides a warm start for the RRT and guides the exploration area, and RRT is built upon Torch layers, which enables the algorithm to explore further area. To circumvent a wider obstacle, it requires more samples, when the number of samples is less than 1200, the tree generating frequency is 10 to 50 Hz, when the number of samples exceeds 2500, the frequency is 3 to 10 Hz.

Compared with Brute Force Branching, Torch addresses the issue of exponential growth and thus covers large area, which is especially important when circumventing wide obstacles. Brute Force Branching can circumvent the obstacle at the width of 30 meters but requires numerously 50000 samples and the tree generation time is over 1 second. Brute Force Branching also encounters heavy burden of computational resource.

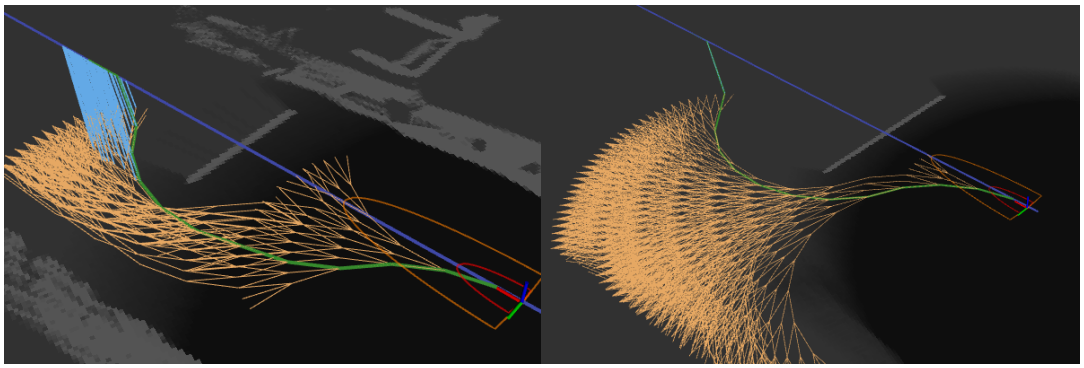


(a) 2-1 **(b)** 2-2

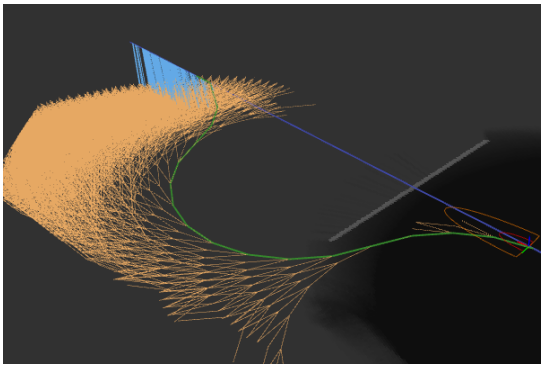


(c) 2-3 **(d)** 2-4

Figure 3.6: Performance of Torch-RRT for scenario 2



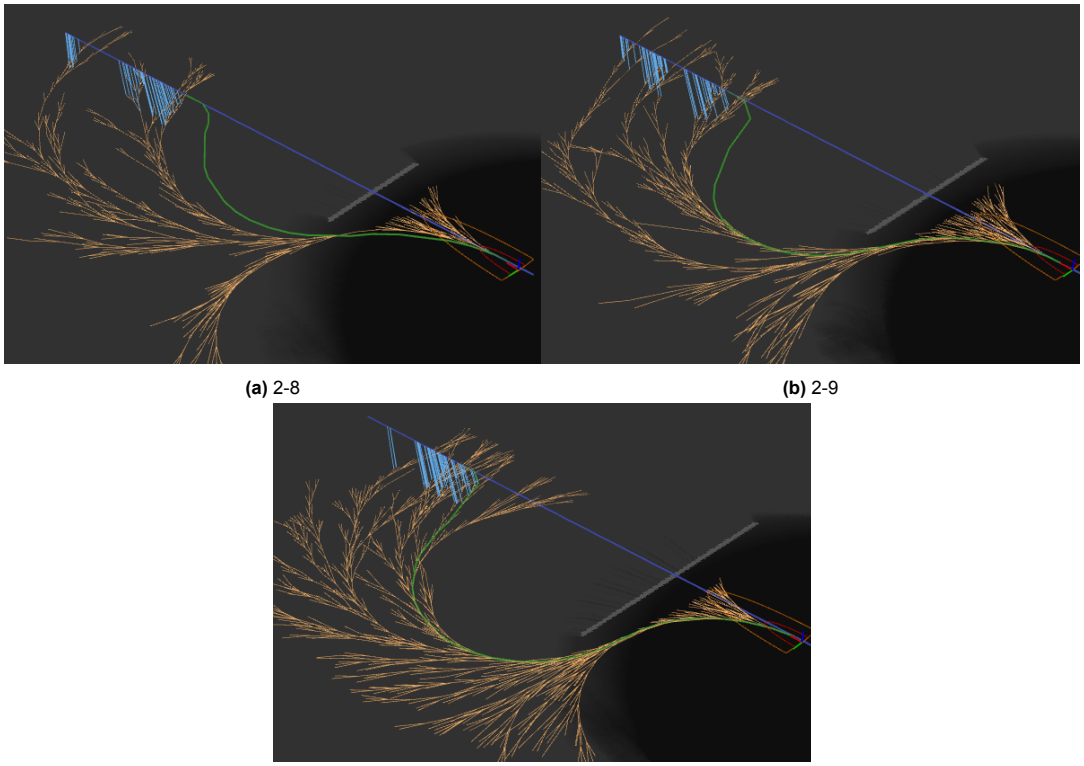
(a) 2-5 **(b)** 2-6



(c) 2-7

Figure 3.7: Performance of Brute Force for scenario 2

Compared with RRT, Torch requires less samples to find a solution and it is also more computationally efficient. What is more, it alleviates the issue of stochastic nature of RRT, since Torch provides a good warm start and guides the exploration space. This can be shown from the fact RRT can not circumvent the obstacle at the width of 40 meters.



(a) 2-8 **(b)** 2-9

(c) 2-10

Figure 3.8: Performance of RRT for scenario 2

3.3.3. Scenario 3

In the third scenario, the algorithms are tested in the simulated Green Village with varying length of long obstacles. The performance of three algorithms including Torch-RRT, Bruteforce Branching and RRT is compared.

Some fixed common parameters are shown in Table 3.5. Among these parameters, the *initialized angle* θ for Torch is set to $15°$, and thus the FoV is $30°$. The checking distance is set to 15. The minimum and maximum lengths of connection are set to 3 and 10, which means the functionality GenerateConnection will create connection edges from the tree to the referenced path with the length ranging from 3 to 10. And the angle between the connection edge and the referenced path is set to $30°$

Parameters	Values
FoV of Torch	$30°$
chekcing distance	15m
min length of connection	3m
max length of connection	10m
connection angle	$30°$

Table 3.5: Fixed parameters for scenario 3

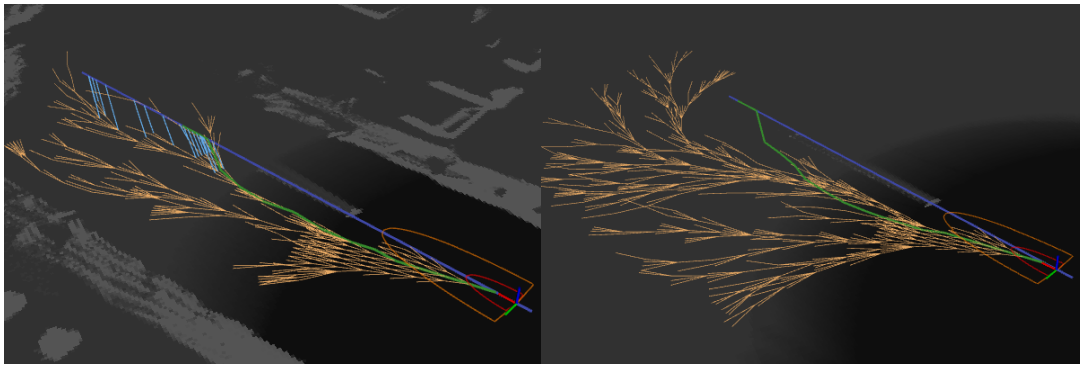
The simulation experiments in scenario 3 mainly tests the ability of different algorithms to circumvent long obstacles. Thus, the performance of the three algorithms with different angle α, edge length L and number of samples N is tested in front of different length obstacles. Two metrics are used to evaluate the performance of different algorithms, including the time of tree generation and the CPU consumption of the software. The parameters and results are shown in Table 3.6.

Scenario	Tree type	l(Obs)	L(Edge)	N(samples)	T(Tree Generation)	CPU
3-1	Torch-RRT	10	2, 3	800	20-60ms	10-19
3-2	Torch-RRT	20	2, 3	1200	20-80ms	10-30
3-3	Torch-RRT	30	2, 3	1500	40-100ms	20-40
3-4	Torch-RRT	40	2, 3	1800	20-120ms	20-50
3-5	Torch-RRT	50	2, 3	18/2000	40-140ms	30-80
3-6	BF	10	3	2000	20-80ms	20-35
3-7	BF	20	3	20/15000	250-400ms	95-100
3-8	RRT	10	3	1000	40-160ms	30-45
3-9	RRT	20	3	1200	40-100ms	25-45
3-10	RRT	30	3	1500	40-100ms	25-60
3-11	RRT	40	3	1800	40-100ms	30-65
3-12	RRT	50	3	2500	60-240ms	40-100

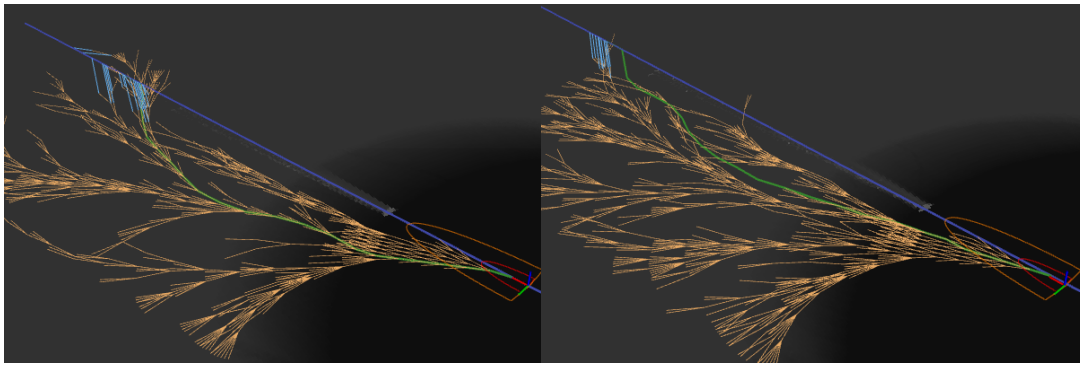
Table 3.6: Results in scenario 3

The screenshots of performance in RViz of the three algorithms in wide obstacle scenario are shown in Figure 3.9, Figure 3.10 and Figure 3.11. In Figure 3.9, the Torch-RRT algorithm is shown to successfully circumvent the long obstacle up to 50 meters length. To circumvent a longer obstacle, it requires more samples, when the number of samples is less than 1200, the tree generating frequency is 10 to 50 Hz, when the number of samples is between 1800 and 2000, the frequency is 7 to 25 Hz.

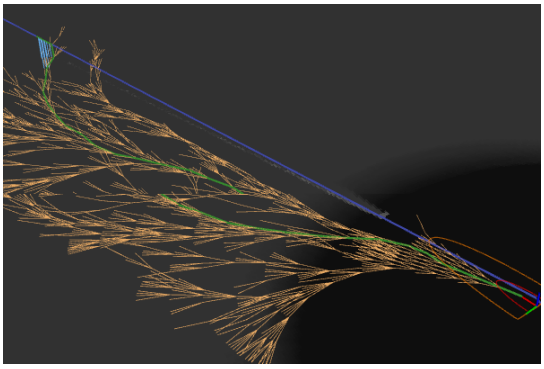
Compared with Brute Force Branching, Torch covers larger area, which helps to circumvent long obstacles. Brute Force Branching can only circumvent the obstacle at the width of 20 meters but requires numerously 15000 samples with a frequency of 2 Hz. Brute Force Branching also encounters heavy burden of computational resource, which is up to 100% CPU resource in one core.



(a) 3-1 **(b)** 3-2

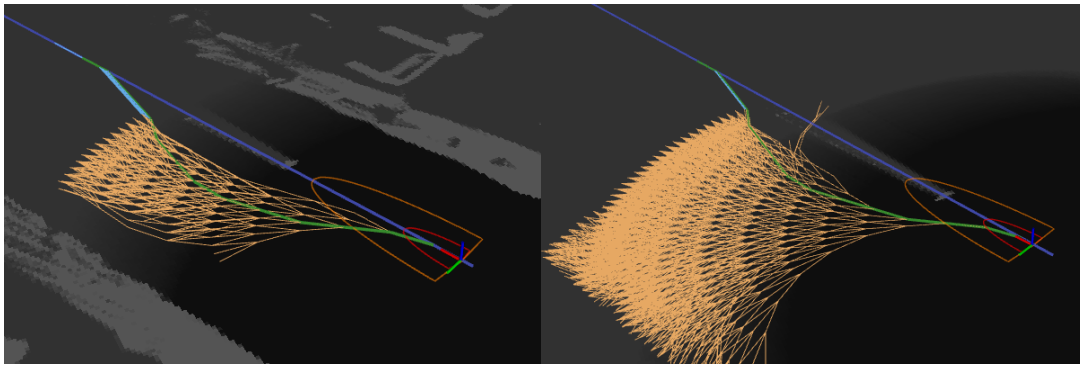


(c) 3-3 **(d)** 3-4



(e) 3-5

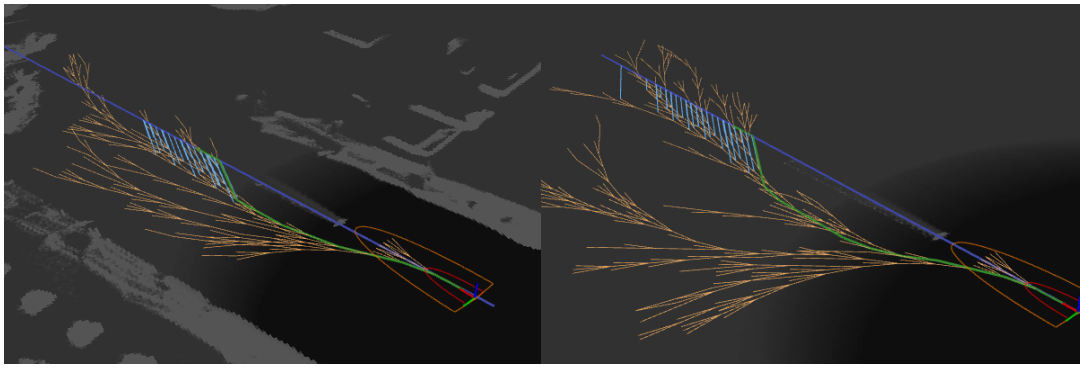
Figure 3.9: Performance of Torch-RRT for scenario 3



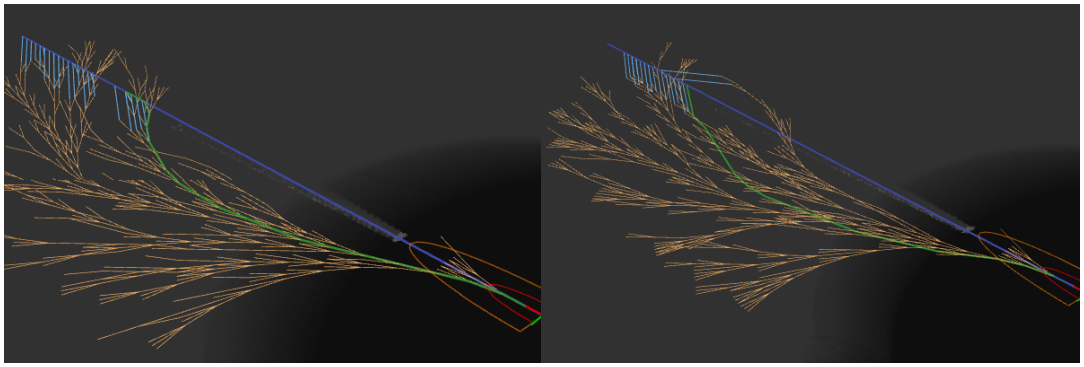
(a) 3-6 **(b)** 3-7

Figure 3.10: Performance of Brute Force for scenario 3

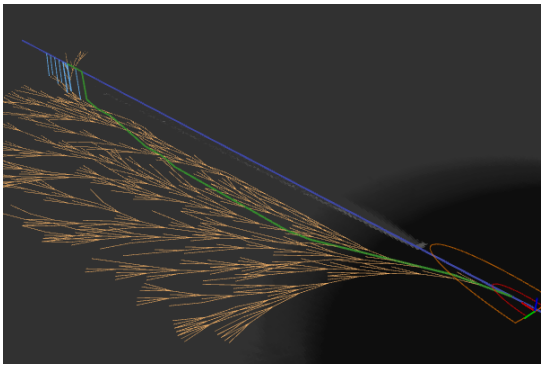
Compared with RRT, Torch requires less samples to find a solution and it is also more computationally efficient. 2000 samples are required for Torch to circumvent 50 meters long obstacle within 7 to 25 Hz, and this outperforms RRT, which requires 2500 samples to circumvent the same obstacle within 5 to 15 Hz.



(a) 3-8 **(b)** 3-9



(c) 3-10 **(d)** 3-11

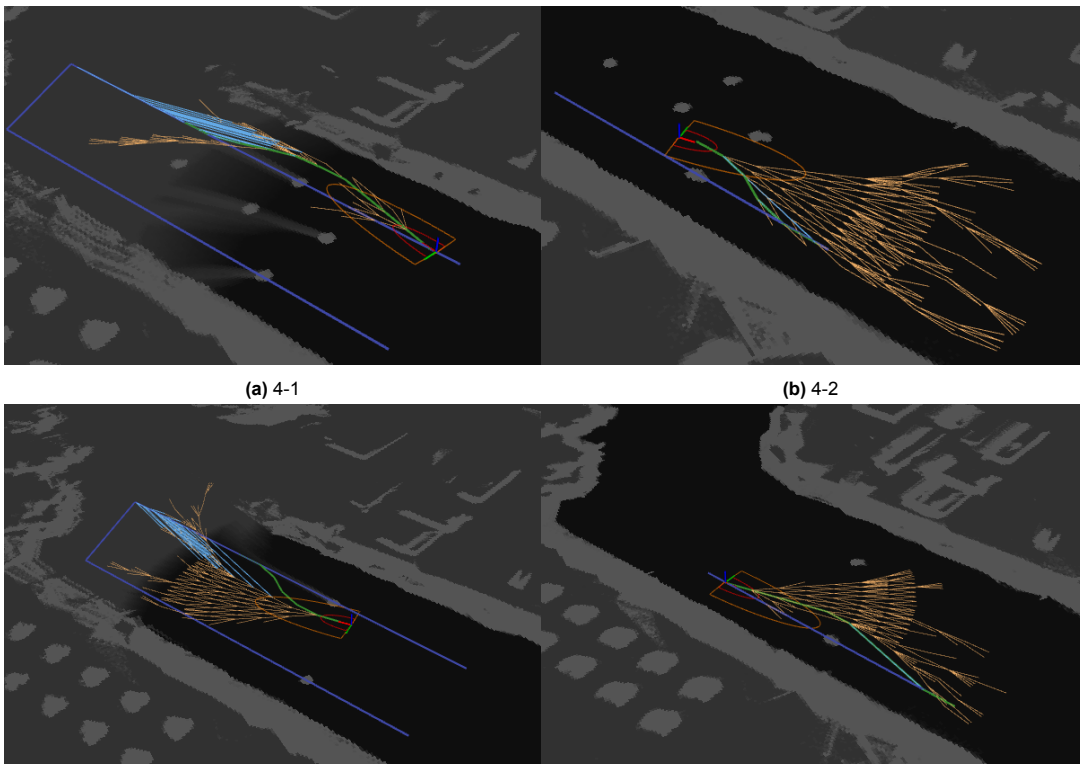


(e) 3-12

Figure 3.11: Performance of RRT for scenario 3

3.3.4. Scenario 4

The proposed Torch-RRT algorithm is also tested in the scenario of multiple obstacles. The performance in RViz is shown in Figure 3.12, where the two figures in the first row contain 5 obstacles and two figures in the second row contain 2 obstacles.



(a) 4-1 **(b)** 4-2

(c) 4-3 **(d)** 4-4

Figure 3.12: Performance of Torch-RRT for scenario 4

The performance of the vessel finishing the task can be shown in Figure 3.13, where the light green curves are the trajectories the vessel has taken for action and the yellow lines are the predefined paths. The trajectories shown are already smoothed after the paths generated by the path planning algorithm.

Figure 3.13: Performance in the simulator

3.3.5. Video Link

The simulation videos of the testing in 4 scenarios can be found in the author's Github repo (see here). However, due to confidential reasons, the repo will be set private after the report is passed and the link will be invalid then.

4

Conclusion

The aim of this intern project is to design an efficient real-time sampling based path planning algorithm for the path tracking of unmanned vessels. The proposed algorithm has a shape of torch and thus be called Torch. It can grow with specified parameters, including FoV, edge angle, edge length, and number of layers. The new structure allows the required samples to grow polynomially with the increase of number of layers, which addresses the exponential growth issue of Brute Force Branching. The algorithm can also be combined with RRT to reach further area, where the torch structure provides a warm start for the RRT and alleviates the stochastic inconsistent issue of RRT, which is beneficial when circumventing wide and long obstacles.

Tests were conducted in the CoppeliaSim Simimulator and ROS environment. Only the static obstacles are considered since the company wants another intern to implement the detection and tracking of dynamics obstacles using Kalman filtering and it is still ongoing and thus the testing performance of dynamic obstacles will be combined in the future. The performance of three algorithms including Torch, Brute Force Branching and RRT is combined in scenarios containing single obstacle, wide obstacle, long obstacle and multiple obstacles. Tests have shown that Torch algorithm can avoid single obstacles with fewer number of samples than the other two algorithms. Torch can be combined with RRT to circumvent 40m wide obstacle and 50m long obstacle. When the number of samples is less than 1200, the tree generating frequency is 10 to 50 Hz, when the number of samples is between 1800 and 2000, the frequency is 7 to 25 Hz. The computational efficiency and CPU consumption of Torch also outperforms the other two algorithms.

In the future, this work might be extended in the following directions. First, the current path planner should be combined with the dynamic obstacle tracking module to update the information about potential dynamic obstacles, which allows the vessels to navigate safely in a more dynamic environments. Second, the mass and drag of the robotic boat may change drastically when transporting goods. Therefore, incorporating an online model identification system and adaptive controllers will be one of the next steps. Third, wave disturbances always exist in natural waters or in fierce weather conditions, which will be addressed in the future controller design.

References

[1] Demcon Unmanned Systems. *Demcon unmanned systems: Overview LinkedIn.* URL: https://www.linkedin.com/company/demcon-unmanned-systems/ (visited on 2022).

[2] John H Reif. "Complexity of the mover's problem and generalizations". In: *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. IEEE Computer Society. 1979, pp. 421–427.

[3] John Canny. *The complexity of robot motion planning*. MIT press, 1988.

[4] Lydia E Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *IEEE transactions on Robotics and Automation* 12.4 (1996), pp. 566–580.

[5] Steven M LaValle and James J Kuffner Jr. "Randomized kinodynamic planning". In: *The international journal of robotics research* 20.5 (2001), pp. 378–400.

[6] Nathan Ratliff et al. "CHOMP: Gradient optimization techniques for efficient motion planning". In: *2009 IEEE International Conference on Robotics and Automation*. IEEE. 2009, pp. 489–494.

[7] John Schulman et al. "Motion planning with sequential convex optimization and convex collision checking". In: *The International Journal of Robotics Research* 33.9 (2014), pp. 1251–1270.

[8] X. Liu, R. Martin Rodriguez, P. Fery, Y. Zhang. *Planning Algorithm for a Quadrotor Drone*. URL: https://www.researchgate.net/publication/358573208_Planning_Algorithm_for_a_Quadrotor_Drone (visited on 2022).

[9] Ltd Coppelia Robotics. *CoppeliaSim*. URL: https://www.coppeliarobotics.com/ (visited on 2022).