Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

A Tree-Based Method for Exploration Using a Car-Like Robot

Scientific Students' Association Report

Author:

Barbara Abonyi-Tóth

Advisor:

Dr. Ákos Nagy

2022

Contents

Kivonat

Kezdetben ismeretlen terek felfedezése és feltérképezése már régóta ismert probléma az autonóm, vezető nélküli járművek világában. Jelen dolgozat célja egy idő-optimális módszer bemutatása, amely alkalmas LIDAR-ral felszerelt autószerű roboton való használatra.

A dolgozat elsőként egy klasszikus, frontier alapú módszert vizsgál az eredeti alakjában, ami egy holonóm roboton való használatra készült. Ezután különböző módosítások kerülnek tárgyalásra, amelyek elősegítik, hogy az algoritmus használható legyen anholonóm járművek esetén is. A változtatások lehetővé teszik, hogy a robot a kiválasztott frontiert a legmegfelelőbb irányból közelíthesse meg.

A dolgozat második fele egy új, fa alapú célpontkiválasztási módszert jár körül, amelynek célja a felfedezéshez szükséges idő csökkentése. Ehhez egy mélységi kereséshez hasonlító módszert alkalmaz a folyamatosan épülő fán. Előbb a fa építésének folyamata kerül bemutatásra, majd a célkiválasztásról olvashatunk részletesen.

A tesztelés és kiértékelés szimulációs környezetben kerül bemutatásra. Előbb egy kisebb, fa szerkezetű folyosó felfedezése lesz a cél, majd egy kiterjedtebb kastély, ami több kört is tartalmaz annak érdekében, hogy a módszereket egy komplexebb környezetben is tesztelni lehessen. A dolgozat a két módszer összehasonlításával zárul.

Abstract

The exploration and mapping of previously unknown space is a long-known problem in the field of autonomous unmanned vehicles. This paper focuses on finding a time-optimal method of exploration to be used on a car-like robot equipped with a LIDAR for scanning its environment.

First, a classical frontier-based exploration method will be reviewed in its original form, which was created to be used on a holonomic robot. Then alterations will be discussed to make it more suitable to use on non-holonomic vehicles, which allow the robot to approach a selected frontier at the most suitable angle.

The second part of this paper proposes a new, tree-based method for goal selection, which is used to reduce the time travelled by the exploring robot. It achieves this by performing a depth-first search-like behaviour on the continuously growing tree. First, the building of the tree will be described then the goal selection method will be shown in detail.

The methods are tested and evaluated in a simulated environment. First, a small corridor will be explored with a tree-like structure, then a larger mansion, which contains several loops to test the exploration methods in a more complex environment. The paper ends with a comparison of the two given methods.

Chapter 1

Introduction

In recent times a great proportion of the most exciting problems currently being researched have to do with autonomous vehicles or robots. There are countless uses for unmanned aerial, ground, surface or underwater vehicles.

One particular area of interest is autonomous exploration. There are numerous uses for exploration not requiring human interaction on the place of exploration. We can think about narrow tunnels where a human could not fit in, but a small robot can, or areas of destruction following disasters or any other causes. In such places, there is often a further danger of debris falling and narrow pathways are also a common occurrence. There can also be scenarios, when the area would be free for human passage, but would be otherwise dangerous - for example filled with smoke or other gas, or affected by nuclear radiation.

In any such case, it is useful to have an inanimate object, like an unmanned robot perform the exploration of the area autonomously, so avoiding endangering human lives. When human interaction is needed, for example when a survivor is found, a map will already be made and their position located.

In light of the above, it comes as no surprise that this is a popular field of research. There is a wide range of literature describing different approaches to exploration. The main three directions include frontier-based methods, random tree-based methods and using neural networks.

The focus of this paper is to review a classical frontier-based method and then propose a new tree-based goal selection method in response to the issues of the first.

The paper is organized as follows. Chapter 2 describes the hardware running the exploration and the necessary program components. The related literature is reviewed in Chapter 3, followed by the description of the original frontier-based exploration method in Chapter 4. In Chapter 5 a new tree-based method is proposed for goal selection during exploration. Chapter 6 conveys simulated experimental results and a comparison of the two methods. The paper ends with a conclusion in Chapter 7.

Chapter 2

Hardware and program components

Though the focus of this paper is the examination of exploration methods, we need to be clear on the responsibilities and function of the other program components that make the exploration possible. But first, let us get familiar with the agent of the exploration, the robot itself.

2.1 Exploring hardware: the robot

The exploration is accomplished using a car-like robot seen in Figure 2.1. It is a modified RC car whit a scale of 1:5, which is used by the department for various related research fields within the VR-car project [2],[1],[23].

Figure 2.1: The robot used for exploration

The robot operates with the following components: an Intel Nuc for central processing, an Nvidia Jetson for graphical calculations and three Raspberry Pi-s for processing sensor data. The robot uses inertial measurement units and encoders to calculate its position and two LIDARs at the bottom to collect information about its environment. All components function using ROS (Robot Operating System) [25].

As the agent is a car-like robot there will be some restrictions on its movement in the environment. A car is a non-holonomic vehicle, which appears as a constraint in path planning. It means that the approaching of goal candidates may take a longer time or even be impossible in some angles, as opposed to most of the robots used in the literature reviewed in Chapter 3. The robot also has a minimal turning radius which will also need to be taken into consideration when planning a path to its destination.

2.2 Program components

The robot uses ROS for the operation and connection of its components. This is a popular operating system in the world of mobile robots and robotic arms. Its building blocks are packages, which are usually responsible for providing a given functionality. The units of execution are called nodes, which communicate with each other by publishing and subscribing to topics.

The main components needed for exploring with a mobile robot can be seen in the following.

2.2.1 The robot and its model

The most important part of any project using a robot is the robot itself. When working in real life this means the sensors, data processing units and actuators of the robot. But as we will see in Chapter 6, this is not always the case. We also need to be able to use the robot for exploration in a simulated environment. This requires the above-mentioned parts and physical features all to be modelled and functional to operate in a simulation. The model must contain the constraints discussed above and be controllable by and also publish the same topics the real robot would. This means that the input of the robot model would be the signals for the actuators and the output would be the data retrieved from the sensors.

2.2.2 Mapping and localization

After the sensors of the robot or its simulated model provide the necessary sensor data the next step is the building of the map. This requires information about the robot's environment measured with the LIDARs at the position of the robot and this position to be determined from the data retrieved from the inertial measurement units and the encoders.

This problem is called Simultaneous Localization and Mapping or SLAM [5] for short. In the project, we use a ROS package pre-made for this. The gmapping package [11] is widely used for performing SLAM using sensor data, such as LIDAR points. The node provides a map as a result, which is in our case a 2D cell grid, in which each cell can contain one of three values: free, occupied or unknown. Part of a map can be seen in Figure 2.2. The free cells are shown in white, the occupied in black and the unknown in grey. The gradient

at the walls is the global costmap provided by move_base while the green dots are the LIDAR measurements.

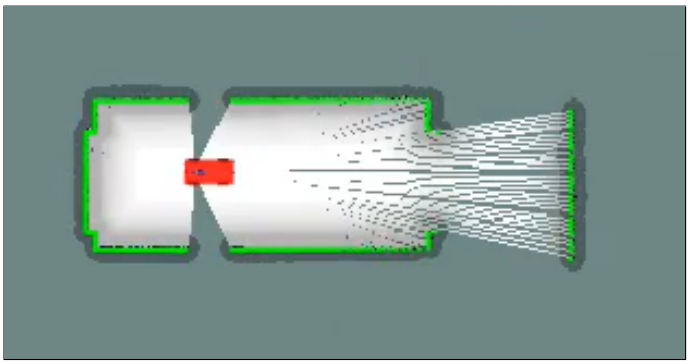


Figure 2.2: The map created by gmapping. The free cells are shown in white, the unknown cells in grey. The LIDAR measurements are represented by the green dots. The gradient colouring by the wall is the global costmap.

2.2.3 The exploration node

The goal of exploration is to explore the whole unknown environment surrounding the robot. To achieve this the robot has to move to get new information about its surroundings. The responsibility of the exploration node is the selection of a goal for the robot, which has the highest potential of revealing unknown areas when approached. The construction of this node is the focus of this paper and as such it will be discussed in detail later in the following chapters.

2.2.4 Global and local path planning

When a goal is selected, the robot needs to move to reach its position. But as there are limitless possibilities to get from one position to another it is important, that the robot follows an optimal path. It is the role of the global path planner to choose a path that satisfies the given pointers of optimality while simultaneously meeting the constraints laid by the physical build of the robot.

The global planner used in this paper is the Hybrid A*. The planner is described in a 2018 Master's Thesis [23] made at the department, which was based on [3]. This type of planner is often used in cases when a non-holonomic robot needs to operate in an unknown environment and replan its path while continuously building an obstacle map at the same time.

The planning consists of two phases. The first phase is to construct a path that satisfies all the kinematic constraints of the robot. The planning is guided by two heuristics. The first plans a path to the origo from its discrete neighbourhood taking into consideration the non-holonomic nature of the robot, but ignoring all obstacles.

The second finds the shortest path to the goal in the obstacle map but ignores the non-holonomic constraints. The heuristics use discretized controls, which are expanded to continuous controls using the Reed-Shepp model [17]. The result is a drivable path, which

is often complex and suboptimal. The second phase of the algorithm uses local optimization and smoothing to arrive at an optimal and drivable path.

The local planner provides the controls necessary to drive along the planned path. These include the velocity of the robot and the steering angles.

The same planners are used with both methods of exploration discussed in Chapters 4 and 5.

2.2.5 The move_base

The move_base [19] is the node responsible for taking the assigned goal and presenting the appropriate controls for moving the robot. It is a part of the navigation_stack and works as an action server to fulfil its purpose.

The exploring node sends the assigned goal to the move_base server through the move_base client, which then runs the global planner to receive a path to the goal. Then the local planner is called which returns the values of the control needed to navigate the robot along the specified path. If any of the planners are unable to come to a solution the move_base server stops the robot.

It also manages the global and local costmaps for the corresponding planners. A costmap is a 2D cell grid similar to the map generated by the gmapping node, but the values of cells are proportionate to the probability of the cell being occupied by an obstacle.

2.2.6 Overview

The cycle of stages of the exploration process are shown in Figure 2.3.

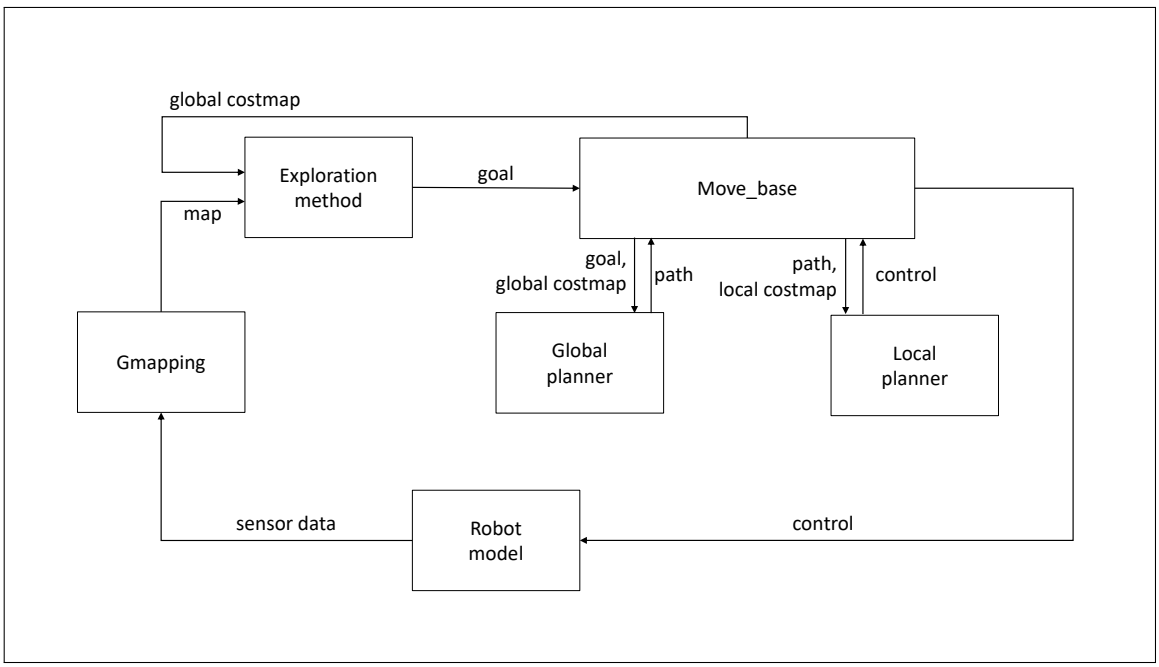


Figure 2.3: The cycle of the exploration process

The exploring node can use either the map generated by the gmapping node or the costmaps managed by move_base to determine the next goal of the exploration. It is then sent to the move_base server which forwards it to the global planner. The received

path is sent to the local planner, which then calculates the necessary controls to follow it. The move_base sends the given control to either the real robot or the simulated model. As the robot moves in the world or in the simulation, the sensors send new data to the gmapping node, which builds the map with a SLAM algorithm. This completes the cycle.

Chapter 3

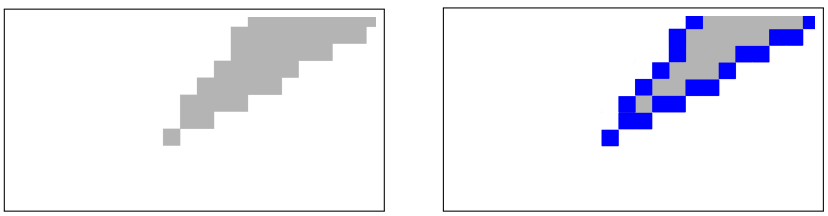
Related work

The problem of autonomous exploration is a popular topic of research papers. Looking at literature related to this problem, we can see three main approaches: frontier-based methods, using randomized trees and using neural networks. In this chapter, each of these categories will be presented through the reviewing of corresponding literature.

3.1 Frontier-based methods

The classical frontier-based approach to exploration originates from the 1997 article of Brian Yamauchi [28]. In contrast to the exploration methods of the time, where either the robot had to follow the walls of the explored room or the obstacles had to be perpendicular, he introduced the concept of frontiers. Defined as the boundaries between known and unknown areas, they are ideal goals of the exploring robot for getting maximum information about its unknown surroundings. When the robot gets to a frontier, new measurements can be taken to expand the known area and push the boundary between the known and unknown areas further into unknown territory.

Yamauchi divides the exploration problem into two parts: frontier detection and navigation to the selected frontier. Imagining the map as an occupation grid of free, occupied and unknown cells, a frontier region can be found as a connected row of unknown cells, all of which have at least one free cell in their immediate neighbourhood. [28] finds the boundaries on the map with a method analogous to edge detection used in computer vision. A frontier can be seen in Figure 3.1, where the frontier points are indicated in blue.



(a) Area for frontier detection **(b)** The points of the frontier

Figure 3.1: Frontier: the boundary between known and unknown regions

After detecting the frontiers on the map at a given time, the ideal goal to pursue is chosen using a greedy algorithm, where the nearest unvisited accessible frontier is considered the

best goal. Then as the robot reaches its destination, new measurements can be taken to expand the map and the next iteration of frontier detection can take place.

This method proved to be better than the other existing methods of the time, and frontier-based exploration soon became a standard in autonomous exploration. Still, it has its drawbacks. The first problem is that the detection of frontiers on a continuously growing map leads to increasing computational costs. The second problem lies with greedy goal selection, as it can lead to a backtracking phenomenon, applying a breadth-first search-like behaviour to the exploration. This, as stated above, is the result of greedy goal selection.

To explain this let the robot stand at crossroads. First, the direction with the highest value frontier will be selected and a path will be planned to it. When the robot gets closer to its goal it gets new information about its environment so the pursued frontier gets pushed back. Meanwhile, the frontiers in the other direction stay the same. The problem arises when the pursued frontier is at this point further away than those in the other direction. When this happens, the robot has to turn back and begins pursuing the selected frontier in the opposite direction. When it gets closer, the same can happen again, resulting in an oscillatory motion, often leaving half-explored rooms behind just to have to come back to them later in a backtracking manner.

Replacing the simple distance-based greedy goal selection algorithm with a more sophisticated selection method can greatly improve the efficiency of frontier-based exploration. The determination of goal candidates and several goal selection methods are discussed in [9]. It compares five goal assignment methods paired with three methods for determining goal candidates. The paper discusses goal assignment methods rather than goal selection, because it focuses on multi-robot exploration, although most of its findings are applicable to single-robot exploration also.

The five assignment methods are the following. The greedy assignment (GA) calculates a utility value for each frontier and chooses the frontier with the highest value as a goal. The iterative assessment (IA) orders each of the $\langle robot, candidate \rangle$ distance pairs and assigns the goals to the robots in that order. The Hungarian Assessment (HA) uses a cost matrix to optimally assign goals to the robots using the aforementioned distances. The Multiple Travelling Salesman Assignment (MA) first clusters the goal candidates and assigns them to each of the robots then explores within the clusters. The Solanas and Garcia Assignment (SGA) also cluster the frontiers and assigns them to the robots then uses a cost computing method described in [9].

The other focus of the paper [9] is the determination of goal candidates. In the first case (AF) all frontiers are selected as goal candidates, which leads to a high computational demand and it is found in [9] that it cannot be used for navigation. The second case uses representatives of free edges (RFE). This way only a few candidates are selected corresponding to the free edges on the map. To achieve this they used a K-means clustering algorithm with a K calculated from the sensor range. The third approach is called complete coverage (CC). This selects the minimum number of points from where all the frontiers can be seen within the sensor range.

After comparing all the candidate determination method and goal assignment pairings the paper [9] concludes, that limiting the number of candidates using simpler assigner methods proves more efficient than using a more complex assigner on a higher number of goal candidates.

3.2 Using trees for exploration

A line of exploration methods was developed as a solution to the first-mentioned problem using random trees to avoid having to use edge detection to detect frontiers. The reviewed literature falls into two categories: using RRT (Rapidly-exploring Random Tree) ([27],[21]) or SRT (Sensor-based Random Tree) for goal selection ([20],[16],[6],[26],[10]).

3.2.1 Rapidly-exploring Random Tree

Using RRT to detect frontiers is beneficial because of the tree's tendency to expand towards unexplored areas, as stated in [27] written by Hassan Umari and Shayok Mukhopadhyay. It also leads to full map coverage, as it ensures completeness. Umari and Mukhopadhyay use two random trees to detect frontiers. One starts from the robot's initial position and expands throughout the exploration, this is called the global detector. The second is reset every time it reaches a frontier point to start expanding again from the current position of the robot, called the local detector. This is meant to ensure the quick detection of frontiers in the near vicinity of the robot, while the global detector finds frontiers further away from the robot and finds its way into small corners, which could be overlooked by an RRT not running for a long enough time. The two trees are shown in Figure 3.2.

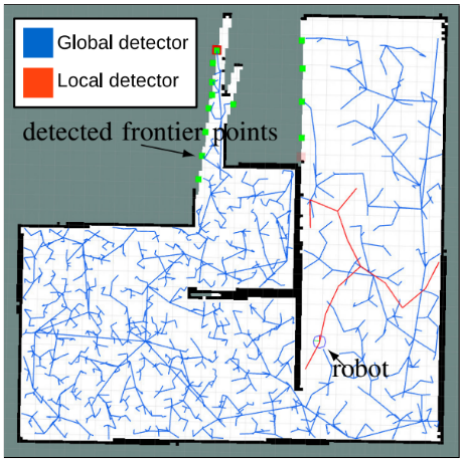


Figure 3.2: Local (red) and global (blue) frontier detectors ([27])

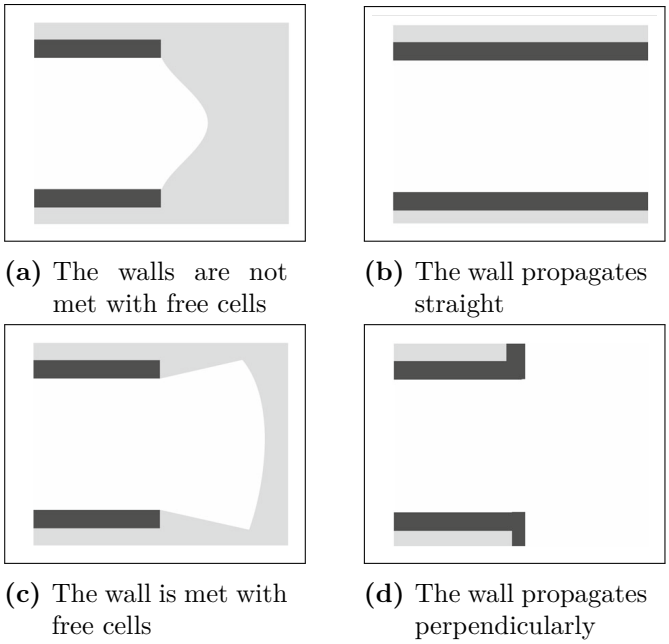
In this paper [27] a randomly generated point becomes a frontier point if it represents an unknown cell or the path from the point to the nearest existing node crosses unknown territory. The exploration process is then divided into two more steps: filtering the detected frontier points and goal allocation to one or more robots.

The filter module is needed because of the high number of frontier points provided by the two frontier detectors, which could lead to visiting overlapping areas and reducing the effectiveness of the exploration. The filter module therefore first clusters the points, saving only the centre points of the clusters. Then it also eliminates any old or invalid frontier points.

The filter's output serves as an input to the task allocator module, which calculates a value called revenue for each of the cluster centres. The supposed information to be gained from visiting the frontier makes it a more desirable goal candidate, while the navigation cost lowers its value. The point with the highest revenue will be the next goal for exploration.

When combined with an estimated information gain RRT can be used to propagate the tree biased towards high-value frontiers, as discussed in [21]. In this case, the RRT is not used to determine frontiers, but to generate an obstacle-free path toward the goal candidates, therefore a separate method of frontier detection is needed. [21] separates the problem into two parts: predicting the information gain of each frontier and using RRT to navigate biased to high information gain frontiers.

For predicting the information gain of a frontier, the paper [21] uses the following method. When the robot travels in a corridor it is surrounded by walls on each side. In this case, examining the area around the frontier separating the known part of the corridor from the unknown can lead to two scenarios. If a wall meets free cells, it is predicted, that at that point the wall changes direction towards the unknown territory (Figures 3.3c and 3.3d). In [21] it is assumed that the walls are perpendicular. When there are no free cells in the line of the wall it is predicted to continue straight (Figures 3.3a and 3.3b). The area of these predictions is proportional to the size of the frontier.



(a) The walls are not met with free cells

(b) The wall propagates straight

(c) The wall is met with free cells

(d) The wall propagates perpendicularly

Figure 3.3: The two ways of propagating frontiers as illustrated by [21]

After predicting the outcome of reaching a frontier the information gain is calculated. For this [21] uses the entropy of the map, which is defined by (3.1), where M is the given costmap, $z_{1:t}$ consists of all observations and actions made by the robot until the given t time and $p_{i,j}$ is the probability of the cell on the i,j coordinates being occupied. The latter can take up three values: $p_{i,j} = 0$, if the cell is free, $p_{i,j} = 1$, if it is occupied and $p_{i,j} = 0.5$ if the state of the cell is unknown.

$$H(M|z_{1:t}) = -\sum_{i,j} p_{i,j} \log p_{i,j} + (1 - p_{i,j}) \log\,(1 - p_{i,j}) \tag{3.1}$$

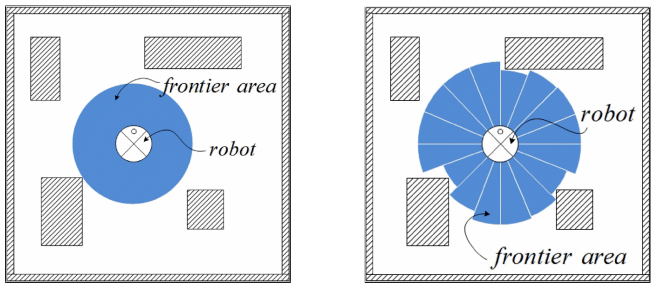
First, the current entropy is calculated from the map without the predictions made. Then the predictions are made one at a time and the posterior entropy is calculated from the

propagated map. The information gain of a frontier will be proportional to the reduction of uncertainty predicted by this method.

The second part of the problem is the decision maker, the role of which is to generate obstacle-free paths in the most relevant destinations. This goal is realized through a biased RRT building. Rather than using random points to propagate the tree [21] uses a method called the Roulette Wheel Selection [12]. Based on the estimated information gain from the previous step the probabilities of selecting the frontier as a new point for the tree are configured, like sections on a roulette wheel, so that larger information gains correspond to larger sections. In all cases, there must be a probability for the method to choose a random point to avoid obstacles in the way. This way the tree will propagate most likely into the direction of frontiers with the highest information gain.

3.2.2 Sensor-Based Random Tree

Another type of randomized exploration method is the SRT. The pure SRT algorithm can be found in [20]. It was developed to eliminate the oscillatory backtracking of goal selection which is the result of the greedy selection algorithm. When the robot's position is added to the tree as a node, a local safe region (LSR) is registered around it, so that there exists a safe path from the robot's position to every point of the safe region. The union of LSRs is the safe region (SR). There are two perspectives introduced in [20] regarding the shape of the LSR. The first discussed perspective is called the Ball-SRT, where the radius of the LSR equals the distance safely approaching the closest obstacle in all directions. This can be seen in Figure 3.4a. Using this method forms a circle around the node and is considered the safer way to generate LSRs. The bolder method is called the Star-SRT, where the radius corresponds to the closest obstacle in its own direction, resulting in a star-like shaped LSR as can be seen in Figure 3.4b.



(a) Ball-shaped LSR **(b)** Star-shaped LSR

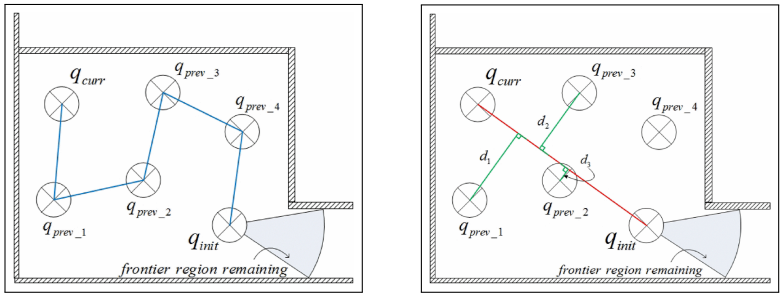
Figure 3.4: The different perspectives of LSR as illustrated by [16]

After the LSR is generated a random direction is selected and a new goal candidate is determined in a way described in [20] in that direction. If the candidate lies too close to the current node or is in the LSR of a previous node, a new direction is randomized. Otherwise, the candidate is selected as a new goal for exploration. When the robot reaches its destination a perception is made using the onboard sensors of the robot and a new node is created with the corresponding LSR.

If the number of attempts to find a valid direction exceeds a given limit, the node is considered explored and the robot backtracks to the previous node and begins searching for valid directions from there. In this way, the completeness of the exploration is ensured and it also serves as an automatic homing method.

This method serves as a built-in depth-first search for exploring an unknown territory, which is preferable to the greedy goal selection algorithms for minimizing the distance travelled by the robot and consequently the time used. On the other hand the backtracking to previous nodes can cause a lot of unnecessary travelling when a simpler route could be found between an explored node and an unexplored node several nodes back.

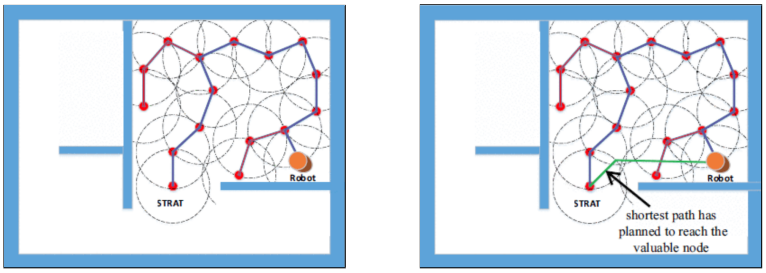
The articles [16] and [6] work on making the backtracking between nodes more efficient. [16] introduces a method called FB-SRT (Frontier-Based-SRT), in which the robot backtracks directly to the last node with frontier edges in its LSR. A frontier edge of the LSR is a boundary between known and unknown regions. This paper, though mentions the Ball-SRT as an option, chooses to work with Star-SRT, as it is more efficient in exploration.



(a) Indirect backtracking **(b)** Direct backtracking

Figure 3.5: Backtracking method of [16]

The method described in this paper is illustrated in Figure 3.5. Figure 3.5a shows the route taken by the robot and the nodes visited while travelling. During the original backtracking method, the robot would travel through the same route in the opposite direction. The direct backtracking method proposed by [16] is shown in Figure 3.5b. First, the first node must be found going back in the tree, which has frontier edges around its LSR. An imaginary line will be drawn between this point and the current position of the robot. Then the distance of the skipped nodes will be measured from this line. These nodes are the candidates for backtracking. The selected candidate will be one that is the closest to the line while also being freely reachable from the current point. In the example given this point will be q_{prev_2}. These steps are repeated until the robot reaches the point with frontier edges. Using this method can eliminate some of the unnecessary travelling during backtracking to previous nodes.



(a) Indirect backtracking **(b)** Direct backtracking

Figure 3.6: Backtracking method of [6]

El-Husseiny et al., 2013 provide a different method for making the backtracking phase of the SRT more efficient ([6]). In the approach discussed the robot backtracks the parent nodes not physically, but through calculations. For each parent node, the information gain is estimated. They propose a ray-casting method for determining which areas of the

node's LSR are valuable for exploration. If the estimated information gain exceeds a given threshold the node becomes the goal of backtracking and the shortest path is planned from the current position of the robot.

Both of these approaches ([16],[6]) were tested in simulated environments and proved to be significantly more efficient than the pure SRT method by eliminating unnecessary travel during backtracking.

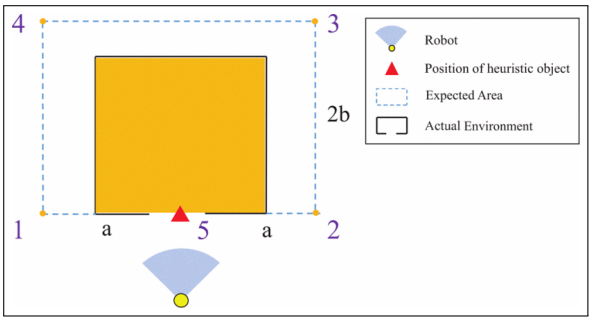


Figure 3.7: Prediction of a room behind a door ([18])

3.3 Using neural networks

Another solution to the oscillatory backtracking problem is proposed in [18]. The focus of the paper is to use computer vision and a lightweight neural network to note if a mapped area is in a sense not complete. Their example for this is a door, behind which a room is suspected. They use the neural network to identify doors in the environment, and when it is found, the robot makes an estimate of the area behind the door in the shape of a rectangular room as can be seen in Figure 3.7. This way it will explore inside the room until it is completely explored, instead of leaving for another direction and having to come back later, so decreasing the time and distance travelled during the exploration.

3.4 Summary

We have seen that numerous methods for the automated exploration problem exist in the literature. The original frontier-based method [28], although often not leading to the most optimal results, tends to be a good foundation to be improved upon. The majority of the other methods fall into two categories: those, that offer a solution to the high computation demands of frontier detection by edge detection, and those that seek to eliminate the backtracking caused by the greedy goal selection method.

The first category, though it is important for real-time use to lower computational times, does not necessarily mean an improvement on the time-efficiency of the exploration. In the second category, however, we can see significant improvements in the time and distance travelled by the robot, which implies, that the main problem for a time-efficient exploration is the backtracking of the greedy algorithm.

We have also seen, that the reviewed literature only uses random trees either to discover frontiers or even for the goal selection and partly path planning for the robot. None of them uses the tree for navigating among frontiers already found.

This paper proposes a method to use trees to deterministically travel through and explore all frontiers. For this the frontiers found with the original edge detector will be organized into a tree structure. The robot performs a depth-first search on the tree while the newly found frontiers are continuously built into it. This method seeks to eliminate unnecessary backtracking and to serve as a time-optimal method of exploration.

Chapter 4

Classical frontier-based exploration

The original work of Yamauchi [28] was used as a base for many of the papers concerned with frontier-based exploration and many methods evolved from it. One of those methods is contained in the ROS package explore_lite. The focus of this chapter will be the review of this method and its completion for easier usage on non-holonomic robots.

4.1 The explore_lite package

The explore_lite package is a ROS (Robot Operating System [25]) package realizing a frontier-based exploration method. Its documentation can be found on the official ROS website [13]. The package was developed in 2016 by Jiří Hörner, a student of the Charles University in Prague [15]. The C code of the package can be freely accessed in its GitHub repository [14].

A frontier is defined as the boundary between known and unknown regions. Usually these are optimal goals of exploration, as there is a high probability that they are reachable and they serve as a good opportunity to explore unknown cells. For this reason, after Yamauchi [28], frontier-based exploration became the most frequented method of autonomous exploration.

The explore_lite package connects to the move_base server with the help of a move_base_client. Through this connection, the exploring node can send the selected goals to the server, which takes care of navigating to the assigned point in the given orientation. Move_base is a part of the ROS package navigation_stack and its purpose is to keep the goal selection, the global and local path planning and the execution of the movements independent from each other. This way any of the listed elements can be replaced without affecting the functionality of the others. The program reads the map data through a similar server-client connection. The client receives the data in the form of a costmap, in which the value of a cell corresponds to the probability that the given cell is occupied by an obstacle.

The program contains two main parts. The outer layer is the exploration itself, which consists of the processing of the map, managing frontiers, goal selection and sending. This makes use of the other part, the role of which is detecting and building the frontiers using the costmap data. The algorithm is described in more detail in Section 4.2.

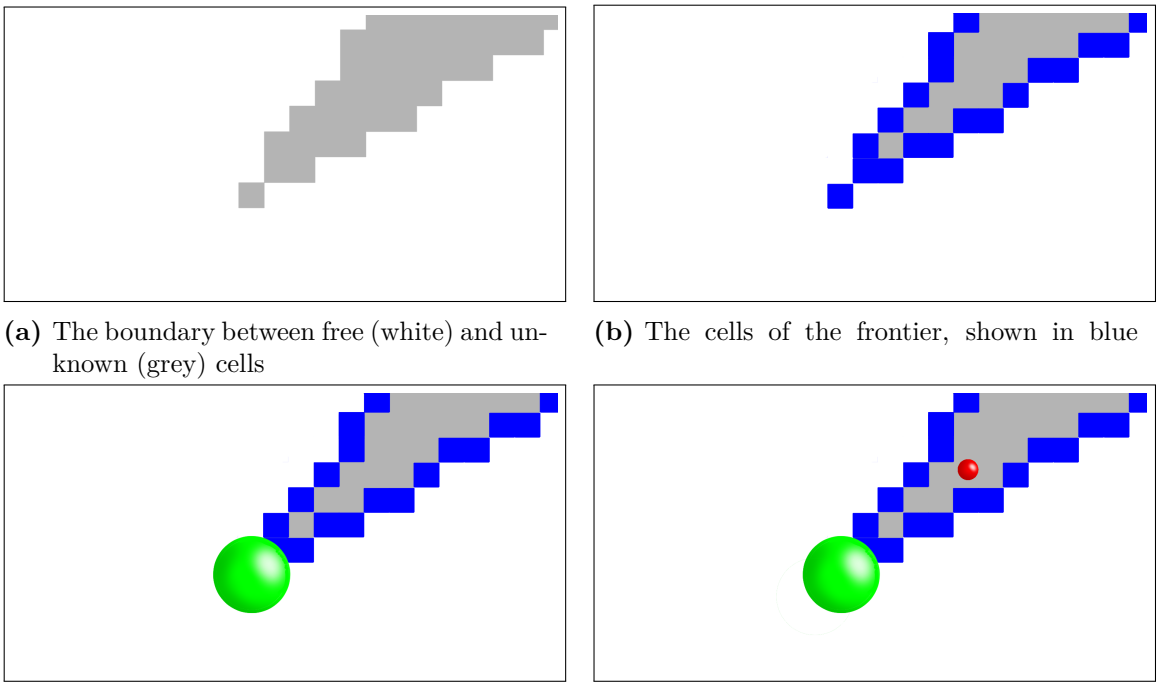
4.2 The algorithm

4.2.1 Frontier detection

The base of this method is the detection, building and management of frontiers. The frontier detection method of this package is based on the ROS package explore [4] with very little modification.

In the code, the detection and building of frontiers are parted into four different problems, which are the following:

- **Identifying new frontier cells.** A cell is a frontier cell if its value on the map is unknown and at least one of its four neighbours is a free cell. For it to be recognized as a new cell it must not have been previously marked as a frontier cell. To determine this the found frontier cells are registered.
- **Building of a frontier.** If at some point during the processing of map data a new frontier cell is discovered the frontier-building method begins. The steps of the process can be seen in Figure 4.1. The firstly discovered cell will be the first frontier point. Then all of its eight neighbours will be tested to find any frontier cells connected to the previously discovered. If a new frontier cell is detected it is registered as one and its neighbours will also be examined. The frontier-building process ends, when no frontier points remain such that would have another frontier point as a neighbour.



(a) The boundary between free (white) and unknown (grey) cells

(b) The cells of the frontier, shown in blue

(c) The closest point to the reference (middle), shown in green

(d) The mean of the frontier points (centroid), shown in red

Figure 4.1: The steps of building a frontier

A frontier contains two special points: the middle and the centroid. The middle is the closest point of the frontier to the position of the robot, while the centroid stands at the average of the frontier points. These can be seen in Figures 4.1c and 4.1d, marked by a green and a red sphere.

The function initializes the centroid x and y coordinates with zero. During the search, when a new frontier cell is discovered the internal variables of the currently built frontier are modified in correspondence with the newly added cell:

  - The size of the frontier is increased by one
  - The x and y coordinates of the centroid are increased by the corresponding coordinates of the new cell
  - The distance of the new cell from the reference is calculated. If the new cell is closer to the position of the robot than the previous minimal distance it takes on the role of the middle cell and the minimal distance is overwritten with the new distance value.

When there are no new frontier cells in the 8-neighbourhood of already processed cells, the building of the frontier comes to its end. Finally, the x and y coordinates of the centroid are divided by the size of the frontier thus giving the mean of the frontier points. The building of the frontier is then finished.

- **Detection of frontiers.** This part is responsible for processing map data to detect frontiers. The search is started from the position of the robot or from a free cell nearest to it. The process uses the costmap from the move_base server.

  First, the nearest free cell to the position of the robot is identified as the initial cell. Similarly to the building of a single frontier this cell is registered as visited and its four neighbours will be examined. The free cells among them will be registered and examined similarly. Unknown cells will be tested for being new frontier cells in the way described above. If a point is found to be a new frontier cell, the building of a new frontier will be started with that point as the initial point. When the building of the frontier ends, the search for new frontiers continues until all free cells become visited and all frontiers are detected.

  If there are no unexamined free cells remaining the algorithm sets the cost of all detected frontiers and so the detection of frontiers ends.

- **Calculating the cost of a frontier.** This function calculates the cost of a frontier as the weighted sum of the size of the frontier and its distance from the position of the robot.

4.2.2 The exploration

The exploration uses the frontiers found in the way described in the previous chapter. The function containing the main code is called at given time intervals. First, it gets the position of the robot from the costmap_client and it starts the frontier detection process from this point and gets in return the array of the frontiers sorted by their cost. From there the algorithm selects those, which are not on the continually expanding blacklist of exploration.

The blacklist contains points which are unreachable goals of exploration. A point gets blacklisted either if no valid path can be planned to it or the robot cannot get closer to it in a given time frame. To determine this the distance of the robot from the current goal is calculated and registered in each iteration. When the distance does not decrease for a time the goal gets blacklisted.

Those frontiers, the centroids of which are on the blacklist, are filtered out from the list of detected frontiers to avoid getting the robot stuck. Then the frontier with the lowest

cost is selected as the next goal of exploration. Since the frontiers were previously sorted by their cost this will always be the first element of the filtered list.

Then the centroid of the selected frontier is sent to the move_base server as a new goal. Its position will equal the coordinates of the pursued point and its orientation is originally a constant vector pointing in the positive x direction.

4.3 Additions for non-holonomic exploration

The original explore_lite package was created for and tested on omnidirectional, circle-shaped robots with sensors that could take measurements in every direction with the same accuracy. For these reasons, there was no need for them to pay attention to the direction of the frontiers or the robot, hence the constant orientation of the goal in a given direction. The robot used in this paper, however, is a car-like robot, which creates a non-holonomic constraint for its movement. A constant direction for the pursued goal is not optimal in our case especially in narrow hallways - which are frequent elements of indoor exploration. Also one of the LIDAR configurations used does not make measurements in the whole 360° around the robot leaving the area behind it unseen, which makes it necessary to approach the frontiers from an ideal angle for maximized information gain.

This requires two parts of the problem to be solved: determining the orientation of a frontier and devising the ideal angle in which to approach the selected frontier.

4.3.1 A change in the blacklisting method

The blacklisting method was changed to only blacklist frontiers if the global path planner cannot make an acceptable path toward it, as waiting for the timeout often led to blacklisting reachable frontiers if in a section of the path the robot had to move away from the goal. On the other hand, waiting for the timeout when the planner cannot make a path is unnecessary and is not time-optimal.

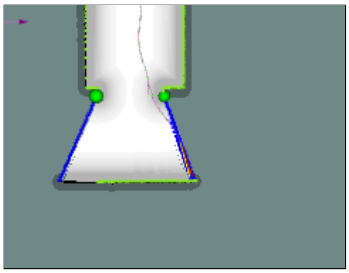
4.3.2 The orientation of a frontier

Two methods were tried to determine the orientation of a frontier. The first idea was to fit a linear line on the points of the frontiers with linear regression using LS (Least Square) approximation [24]. This did not result in the desired effect as the estimated line was often perpendicular to the direction of the frontier rather than parallel. This usually happened when the points did not align in a linear line This is illustrated in Figure 4.2b.

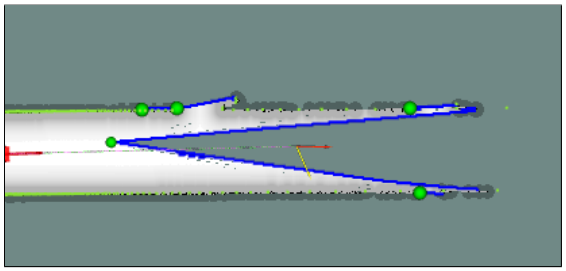
The other approach was to simply use the vector pointing from the middle to the centroid point as the direction of the frontier (see (4.1), where $\mathbf{v}_{\text{frontier}}$ is the orientation of the frontier and $\mathbf{x_c}$ and $\mathbf{x_m}$ are the centroid and middle points). This conveyed our intentions well enough and provided a way to maximize the exploration of the unknown area if the LIDAR cannot measure behind the car.

$$\mathbf{v}_{\text{frontier}} = \mathbf{x_c} - \mathbf{x_m} \tag{4.1}$$

$$\alpha = \text{atan2}\left(\mathbf{v}_{\text{frontier}}\left[y\right], \mathbf{v}_{\text{frontier}}\left[x\right]\right) \tag{4.2}$$



(a) The approximated direction is right when the points align in a linear line



(b) The approximated direction is wrong when the points diverge from a linear line

Figure 4.2: The orientation of a frontier using LS estimation (yellow) and middle-to-centroid vector (red)

In the code, the orientation of a goal is described using quaternions so it is necessary to transform the orientation which is calculated in degrees. In the case of an α angle rotation around the z-axis, the quaternion can be calculated as shown in (4.3).

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \sin\frac{\alpha}{2} \\ \cos\frac{\alpha}{2} \end{bmatrix} \tag{4.3}$$

4.3.3 The ideal angle of approach

We have successfully determined the orientation of the frontier in Section 4.3.2. However, this does not mean that approaching a frontier parallel to this direction will always be optimal. To maximize the information gain of approaching a frontier some frontiers have to be approached perpendicular to their original direction. This way we can differentiate between two types of frontiers which can be seen in Figure 4.3.

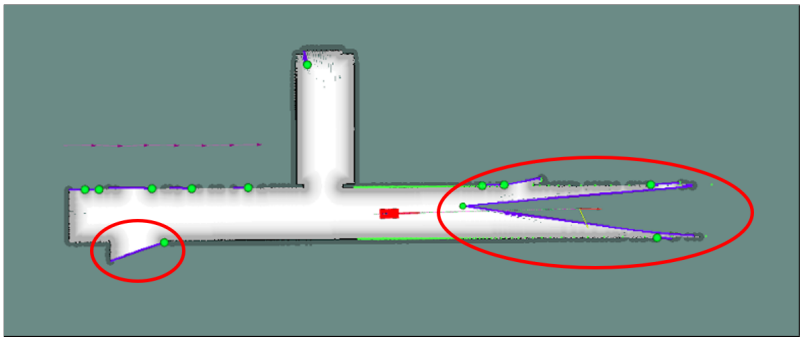


Figure 4.3: The different types of frontiers regarding the ideal direction of approach

The first type is the V-shaped frontier, which usually marks straight hallways with a free line of sight, where the legs of the letter V correspond to the farthest point of the wall as seen by the LIDAR. This shape is created by the gmapping node. When a LIDAR beam does not reach an obstacle the area covered by it is not marked as free from the car to

the end of the measurement range of the LIDAR but rather as unknown territory. This type is optimally approached in the original direction of the frontier and will be called henceforth frontier type A. The other type is the linear frontier. These represent borders where the LIDAR could not see directly into the given area due to an obstacle being in the way. These are usually found when exploring doors or the beginning of hallways. These frontiers are best approached perpendicular to their orientation facing the unknown area and they will be called frontier type B.

4.3.3.1 Differentiating between frontier types

To make a distinction between the two types of frontiers we will approximate the points of the frontiers with a linear using LS estimation. For this, linear regression will be calculated for the frontier points as measurement data following the method described in [24]. We will use the measurement model seen in (4.4) and (4.5).

$$y_n = a_0 + a_1 u_n + w_n \tag{4.4}$$

$$\mathbf{z} = \mathbf{U}\mathbf{a} + \mathbf{w} \tag{4.5}$$

The equation (4.5) is filled with measurement data in (4.6), where y_n will be the y and u_n the x coordinate of a frontier point and $\mathbf{a}$ the parameters of the linear estimation. $\mathbf{W}$ represents the error in our measurement.

$$\mathbf{z} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & u_0 \\ 1 & u_1 \\ \vdots & \vdots \\ 1 & u_{N-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} + \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N-1} \end{bmatrix} \tag{4.6}$$

$$\mathbf{U} = \begin{bmatrix} 1 & u_0 \\ 1 & u_1 \\ \vdots & \vdots \\ 1 & u_{N-1} \end{bmatrix} \tag{4.7}$$

It is stated in [24] that the estimation of the parameters can be calculated as seen in (4.8). After some transformation, we get the formula used to calculate the parameters of the linear line (4.10).

$$\hat{\mathbf{a}} = \left[\mathbf{U}^T\mathbf{U}\right]^{-1}\mathbf{U}^T\mathbf{z} \tag{4.8}$$

$$\left[\mathbf{U}^T\mathbf{U}\right] = \begin{bmatrix} N & \sum_{n=0}^{N-1} u_n \\ \sum_{n=0}^{N-1} u_n & \sum_{n=0}^{N-1} u_n^2 \end{bmatrix}, \quad \mathbf{U}^T\mathbf{z} = \begin{bmatrix} \sum_{n=0}^{N-1} y_n \\ \sum_{n=0}^{N-1} u_n y_n \end{bmatrix} \tag{4.9}$$

$$\begin{bmatrix}\hat{a}_0\\ \hat{a}_1\end{bmatrix} = \frac{1}{\frac{1}{N}\sum_{n=0}^{N-1} u_n^2 - \left(\frac{1}{N}\sum_{n=0}^{N-1} u_n\right)^2} \begin{bmatrix}\frac{1}{N}\sum_{n=0}^{N-1} u_n^2 & -\frac{1}{N}\sum_{n=0}^{N-1} u_n\\ -\frac{1}{N}\sum_{n=0}^{N-1} u_n & 1\end{bmatrix} \begin{bmatrix}\frac{1}{N}\sum_{n=0}^{N-1} y_n\\ \frac{1}{N}\sum_{n=0}^{N-1} u_n y_n\end{bmatrix} \tag{4.10}$$

The value used to differentiate between the frontier types will be the least square error (LSE) itself, as it describes the points' divergence from the linear line well. The LS error value can be calculated with the formula (4.11), where x_n and y_n are the coordinates of the nth frontier point.

$$\text{LSE} = \frac{1}{N}\sum_{n=0}^{N-1} \left(y_n - (a_0 + a_1 \cdot x_n)\right)^2 \tag{4.11}$$

Though the direction of the linear line resulting from this method is not always useful we can make use of the LS error of the estimation to note a difference between the two types of frontiers. Since points of the type B frontier generally fall in one linear line the LS error for these frontiers will be low. In this case, the estimated line falls in the same line as the points of the frontier (Figure 4.2a). On the other hand, type A frontiers tend to diverge from one line and therefore the LS error in their case will be much higher. In this case, the estimated linear does not fall in the direction of the frontier.

4.3.3.2 Classification of frontiers

Type B frontiers are in need of further diversification regarding the ideal angle of approach. For this we have to look at two points, one on each side of the linear, the value of which identifies four subtypes:

I. Known cell over it, unknown cell under it

II. Known cell under it, unknown cell over it

III. Known cells both over and under it

IV. Unknown cells both over and under it

To select the points to examine we have to determine the normal vector of the linear. This can easily be done by looking at the LS estimated linear, for the second parameter (a_1) gives the slope of the line. Using this the direction vector can be calculated as seen in (4.12).

$$\mathbf{v} = \begin{bmatrix}x\\ y\end{bmatrix} = \begin{bmatrix}1\\ a_1\end{bmatrix} \tag{4.12}$$

Rotating the direction vector by 90° we get the normal vector of the line (4.13).

$$\mathbf{n} = \begin{bmatrix}x\\ y\end{bmatrix} = \begin{bmatrix}-a_1\\ 1\end{bmatrix} \tag{4.13}$$

Using this we can define the two points on the two sides of the line. They are calculated as the centroid point offset with the positive and the negative normal vector. For this, we are stretching both vectors by ten times the resolution of the costmap so that the points will be far enough from the points of the frontier. Equations (4.14) and (4.15) show the calculation of the two points, where $\mathbf{x_c}$ is the position of the centroid.

$$\mathbf{p} = \mathbf{x_c} + 10 \cdot \text{res} \cdot \mathbf{n} \tag{4.14}$$

$$\mathbf{q} = \mathbf{x_c} - 10 \cdot \text{res} \cdot \mathbf{n} \tag{4.15}$$

Computed like this $\mathbf{p}$ will always be in the positive, while $\mathbf{q}$ will be in the negative y direction.

The orientation of the frontier ($\mathbf{v}_{\text{frontier}}$, (4.1)) needs to be rotated to get the best direction of approach only if the cell on one side is unknown and the one on the other side is known. To decide the direction of the rotation with respect to the direction of the frontier we have to take into account the four cases depicted in Figure 4.4.

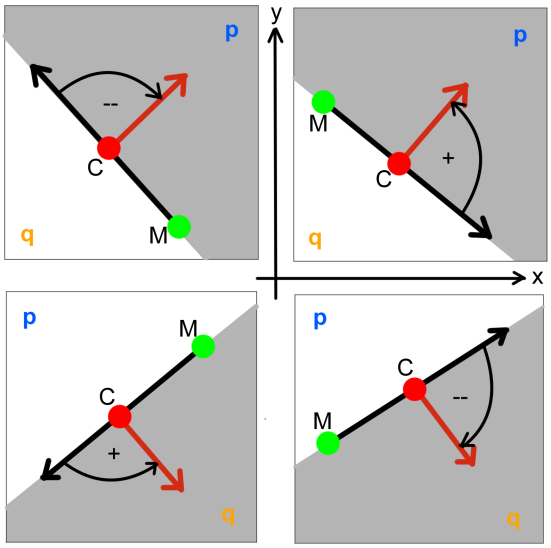


Figure 4.4: The four cases of rotating the direction of the approach. The black arrow shows the orientation of the frontier (pointing from the middle to the centroid) and the red arrow shows the best direction of approach.

In light of this, we have to rotate the orientation vector of the frontier in the positive direction, meaning we have to add $\pi/2$ radians in the following cases:

- the point over the line ($\mathbf{p}$) is known, the point under the line ($\mathbf{q}$) is unknown and the x coordinate of the centroid is smaller than the x coordinate of the middle
- the point over the line ($\mathbf{p}$) is unknown, the point under the line ($\mathbf{q}$) is known and the x coordinate of the centroid is bigger than the x coordinate of the middle

On the other hand, the vector needs to be rotated in the negative direction, meaning we have to subtract $\pi/2$ radians in the following cases:

- the point over the line (**p**) is known, the point under the line (**q**) is unknown and the x coordinate of the centroid is bigger than the x coordinate of the middle
- the point over the line (**p**) is unknown, the point under the line (**q**) is known and the x coordinate of the centroid is smaller than the x coordinate of the middle

In favour of clarity let us see these conditions in a tabular form in Table 4.1.

Type B frontier	**p** known, **q** unknown	**p** known, **q** unknown
centroid.x < middle.x	$+\frac{\pi}{2}$	$-\frac{\pi}{2}$
centroid.x > middle.x	$-\frac{\pi}{2}$	$+\frac{\pi}{2}$
Type A frontier	no need for rotation	

Table 4.1: Rules for rotating the direction of frontiers

In the case of type A frontiers, the direction of approach is the orientation of the frontier, there is no need for any rotation.

4.3.3.3 Visualization of frontier classes

The four categories mentioned above are identified when computing the orientation of the frontier. Then, when the frontiers are visualized the original blue colour of frontier points changes according to the type of the frontier as seen in Table 4.2.

p	**q**	type	Colour of frontier points
-	-	A	green
known	unknown	B/I.	yellow
unknown	known	B/II.	purple
known	known	B/III.	turquoise
unknown	unknown	B/IV.	blue

Table 4.2: Visualization of frontiers

Figure 4.5 shows the colouring in the simulated environment.

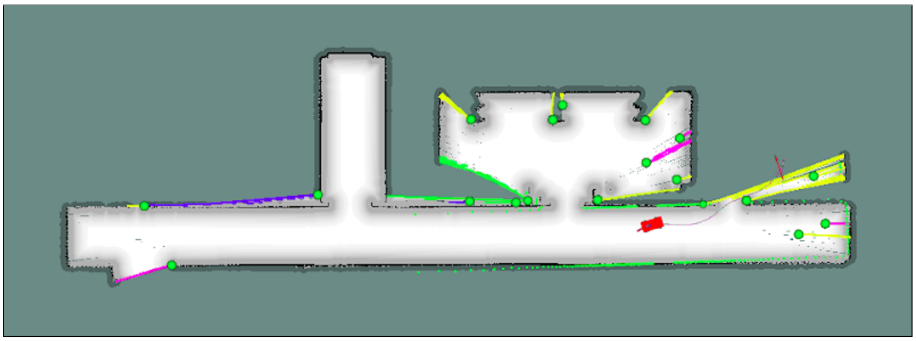


Figure 4.5: Frontier classes visualized with different colours: green - type A (V-shaped) frontier, yellow - type B/I., purple - type B/II., blue - type B/IV. frontiers. There are no instances of turquoise - type B/III. frontiers, as they generally do not appear.

Chapter 5

The tree-based method of exploration

After seeing the pitfalls of the greedy goal selection of explore_lite, this chapter's goal is to introduce a new method for goal selection. This includes organizing the detected frontiers into a tree structure and proposing a selection algorithm using the tree as a base. But first, let us understand the reason behind the need to change the greedy strategy for goal selection.

5.1 Motivation

The original frontier-based method of exploration, though cherished for its simplicity, does not always lead to an optimal trajectory, as we have already seen in Chapter 3. One of the problems mentioned was the result of greedy goal selection.

The cost of a frontier consists of its distance from the robot and the expected information gained from exploring it. Often the latter can be a very similar value among frontiers, especially in symmetrical arrangements. This causes the distance of the frontier to occupy a greater role in the goal-selection process.

When the greedy goal selection considers mostly the distance of the frontier as its cost the following scenario can take place: the robot starts pursuing the closest frontier. As it gets nearer, the sensors on board of the robot gain new information about the environment around the approached frontier, which leads to the pushing back of the frontier line. This way the distance between the robot and the pursued frontier increases, while the other frontiers stay in the same place or get pushed back a smaller distance. When this happens, the greedy selection algorithm often assigns another frontier to pursue, which previously had a greater distance from the robot.

For some time the same can happen repeatedly as the robot approaches the newly assigned frontiers, which leads to an oscillation in the trajectory. The phenomenon is illustrated in Figure 5.1. This not only takes a lot of unnecessary time but by visiting the same places over and over again we get very little new information about the robot's environment.

The literature reviewed in Chapter 3 offers various solutions to this problem, most of which use tree structures to get rid of greedy goal selection. The used trees, however, being RRT (Rapidly-exploring Random Tree) or SRT (Sensor-based Random Tree), both contain a random factor in the exploration.

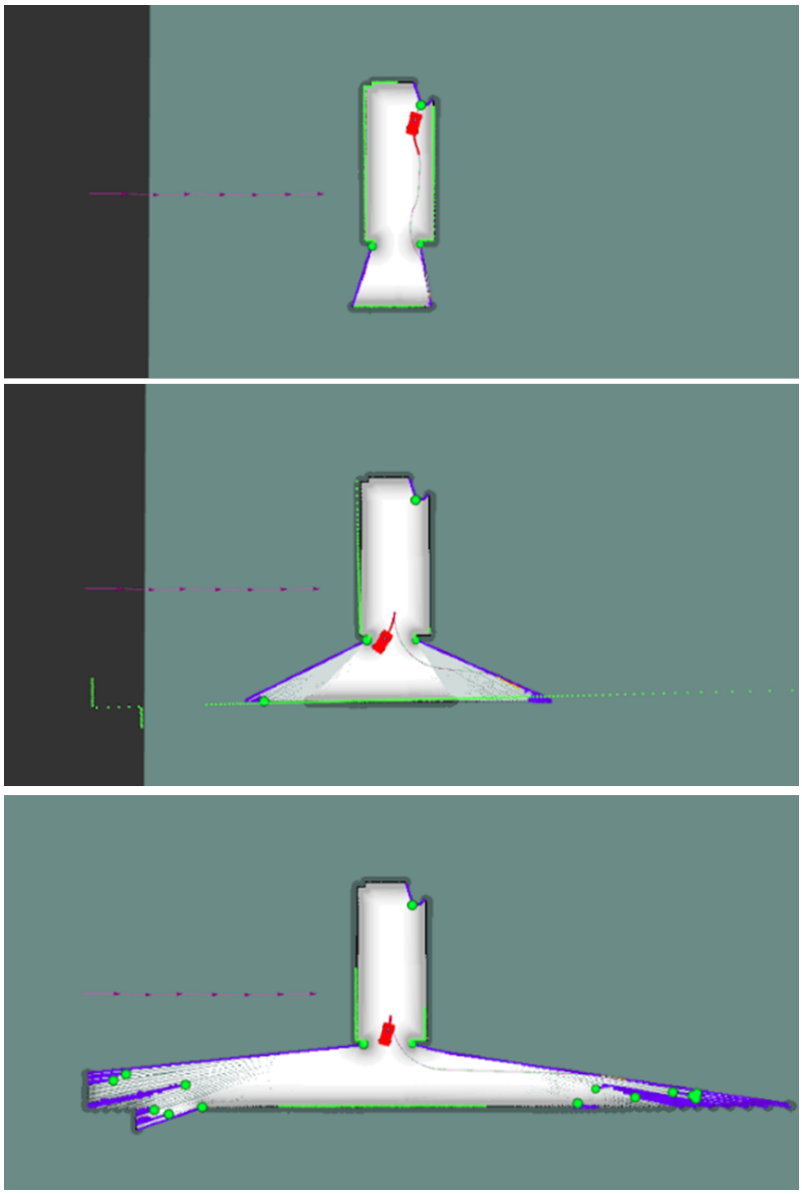


Figure 5.1: The robot is exploring in both directions due to the oscillation of the goal selection, while it returns to the the middle with every change of direction

In this chapter, we will propose a deterministic tree-building and goal selection method for exploration to avoid both the oscillatory problem of the greedy goal selection and the probabilistic nature of the reviewed tree-based exploration methods.

5.2 Overview of the algorithm

The backbone of the proposed algorithm is the tree. But unlike the methods reviewed in Chapter 3, the leaves of the tree are not determined randomly. In each iteration, the representatives of the newly detected frontiers will be built into the tree. This prevents the tree from growing in dead-end directions and creates a tree that better reflects the topology of the environment. The representatives are selected similarly to what [9] describes as RFE

(representatives of free edges). But instead of a K-means clustering, our algorithm uses a different clustering method which will be discussed later in this chapter.

The tree structure provides a base for the goal selection method. An algorithm resembling a depth-first search is used on the continually expanding tree with the meaning of following the same branch as long as it still has any unexplored leaves. When that is not the case and the branch is explored the closest branch will be selected and followed until explored. The detailed explanation of this method can be found in Section 5.4.

The steps of the algorithm are the following:

0. Initialization of the tree
1. Frontier detection
2. Clustering of frontier points
3. Filtering blacklisted frontiers
4. Branching from the tree
5. Goal selection in the tree

5.3 Building the tree

5.3.1 Frontier detection

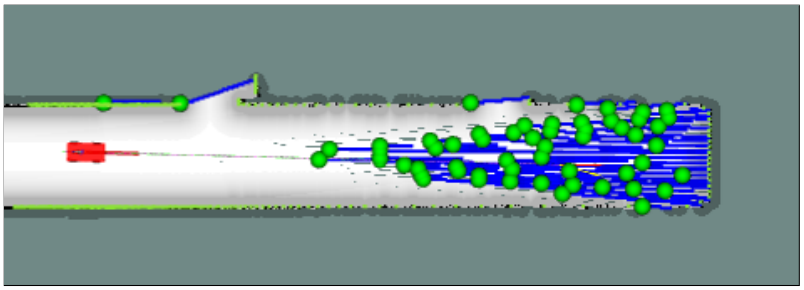
The algorithm is based on the frontier detection method reviewed in Section 4.2.1. Unlike the original, in this case, the points of the frontiers are not managed separately for each of the frontiers, rather they form a collective set, which can be clustered later. Coincidentally the calculation of the frontier's parameters - its centroid, middle, size and orientation - is not taking place during frontier detection but after the clusterization process instead.

5.3.2 The clustering of frontiers

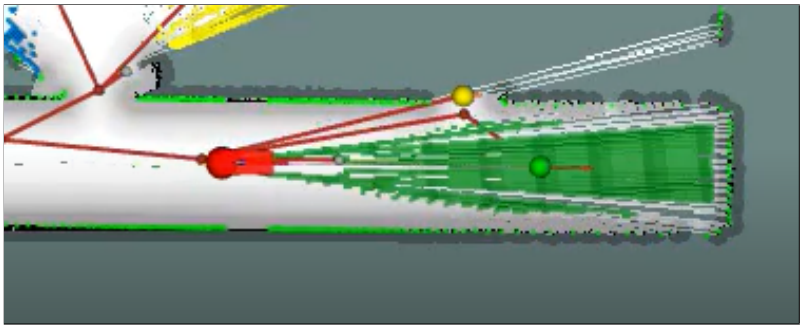
The frontiers, when detected separately, consist of points that are connected to each other, meaning that each point is in the 8-neighbourhood of another frontier point. On the other hand, those frontier points that are not directly connected through their neighbourhood must form a separate frontier, however close they may be to each other. This leads to a high number of frontiers but with minor differences in the area they provide for exploration. Using all frontiers as leaves would clutter the tree without providing a greater information gain. To decrease the frontiers' number to a reasonable amount with the highest possible information gain on disjunct areas the frontier points need to be clustered. The difference can be seen in Figure 5.2.

The clustering method found in the literature [9] is the K-means algorithm, for which the K is determined using the range of the robot's sensor. This holds two disadvantages. The first is that the number of clusters needs to be specified ahead of time. The other problem is that the shape of the clusters is not very flexible and the connectedness of a set of points does not guarantee that they will belong to the same cluster.

For these reasons the method used to cluster the frontier points in our implementation is the DBSCAN (Density-Based Spatial Clustering of Applications With Noise) algorithm



(a) The area without clustering: many frontiers are registered



(b) The area after clustering: only one cluster is registered

Figure 5.2: The same area with and without clustering the frontier points. The clustering leads to fewer goal candidates which correspond better with the possible directions of the exploration.

[8]. This method does not take any input arguments other than the points to be clustered and sorts them into clusters based on their connectedness.

The algorithm uses two parameters: ϵ, which is the distance within a point can be considered connected to another, and $minPts$, which defines a minimum number of points which have to be in the ϵ neighbourhood of a point for it to be considered a *core point*. A point is *directly reachable*, if it is within ϵ distance of a core point, and a q point is *reachable* from a core point if a chain of core points can be formed, in which the neighbouring core points are directly reachable from each other and the q point is directly reachable from the last core point. Any point that is not reachable from any of the core points is considered an *outlier*.

The core points and all points that are reachable from them form a cluster.

The implementation of the DBSCAN algorithm that is used in this paper can be found on GitHub [7]. From this point, the clusters take on the previous role of the frontiers. After clustering the frontier points, the frontier parameters are calculated using the points clustered together. The result of the clusterization is illustrated in Figure 5.3 where the different clusters are indicated with different colours.

Since the points of the clusters are coloured according to their corresponding cluster the frontier types based on the ideal angle of approach are from now on visualized as the colour of the sphere representing the centroid point of a cluster, as can be seen in Figure 5.4. The colour of the sphere is determined as seen before (Table 4.2).

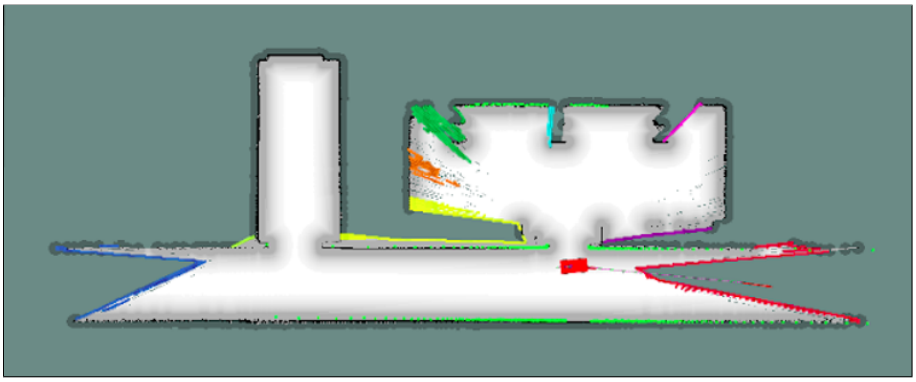


Figure 5.3: The result of the DBSCAN clusterization. Every cluster is shown in a different colour.

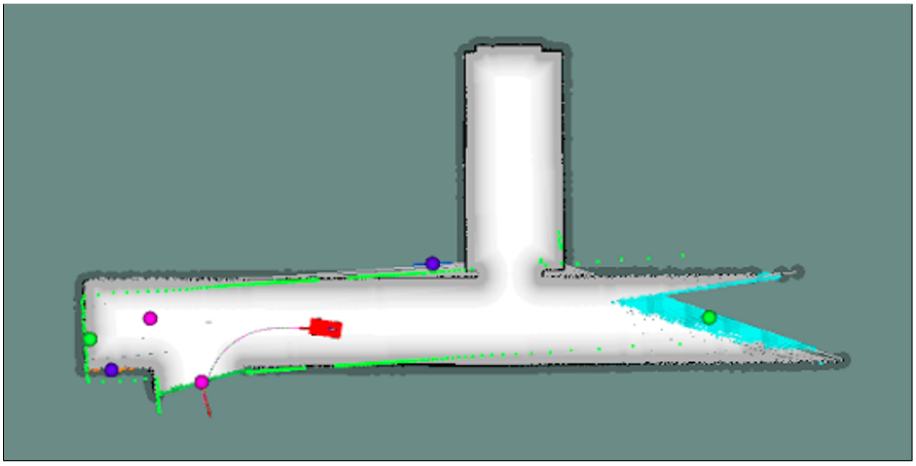


Figure 5.4: The visualization of the clusters alongside the frontier classes. The colour of the spheres corresponds to the represented frontier class: green - type A (V-shaped) frontier, yellow - type B/I., purple - type B/II., blue - type B/IV. frontiers.

5.3.3 Branching from the tree

When the frontier points are clustered and the frontier parameters are set the created clusters are filtered through the blacklist the same way the frontiers were. The non-blacklisted clusters are used for the building of the cluster tree.

The root of the tree is initialized with a cluster which consists of one point, which is the initial position of the robot. From then the new clusters are used to branch out in different directions of exploration.

First, the tree-building method has to identify those clusters which did not move since the last iteration of tree-building. Comparing the leaves of the tree from the last iteration with the newly found clusters three cases can occur:

- A new cluster is present as a leaf in the existing tree. In this case, there is nothing to be done and the leaf is kept as *alive*.
- A leaf is not present among the new clusters. This indicates that the cluster of the leaf is either explored or moved. The leaf is marked as *not alive*.

- No leaf corresponds to the new cluster. These are the clusters that appeared since the last iteration of the tree-building. The new clusters will be marked as *alive*.

In this case, *alive* means that a cluster is presently associated with the given leaf. If a leaf is *not alive*, it means that it no longer represents a frontier.

Following the matching of the leaves and clusters, the appeared clusters need to be connected to the tree. The candidates to be the parents of such a node are the leaves of the previous iteration (old leaves), and their parents. For each of the new clusters, the parent is determined through the following steps.

The valid candidates are selected from the candidates. A candidate is valid if its centroid is on the same side of the robot as the centroid of the appeared cluster so that the appeared cluster could be in the line of sight of the LIDAR of the robot when approaching the parent candidate. This can be checked by considering the angle between the vectors pointing from the position of the robot to the two points in question. If the angle is smaller than 90 degrees, the candidate is valid. According to Pythagoras' Theorem, the candidate has to satisfy the condition shown in (5.1), where $d_{ac,r}$ and $d_{pc,r}$ are the distances of the appeared cluster and the parent candidate from the robot, and $d_{ac,pc}$ is the distance between the appeared cluster and the parent candidate. The possible arrangements of points are illustrated in Figure 5.5.

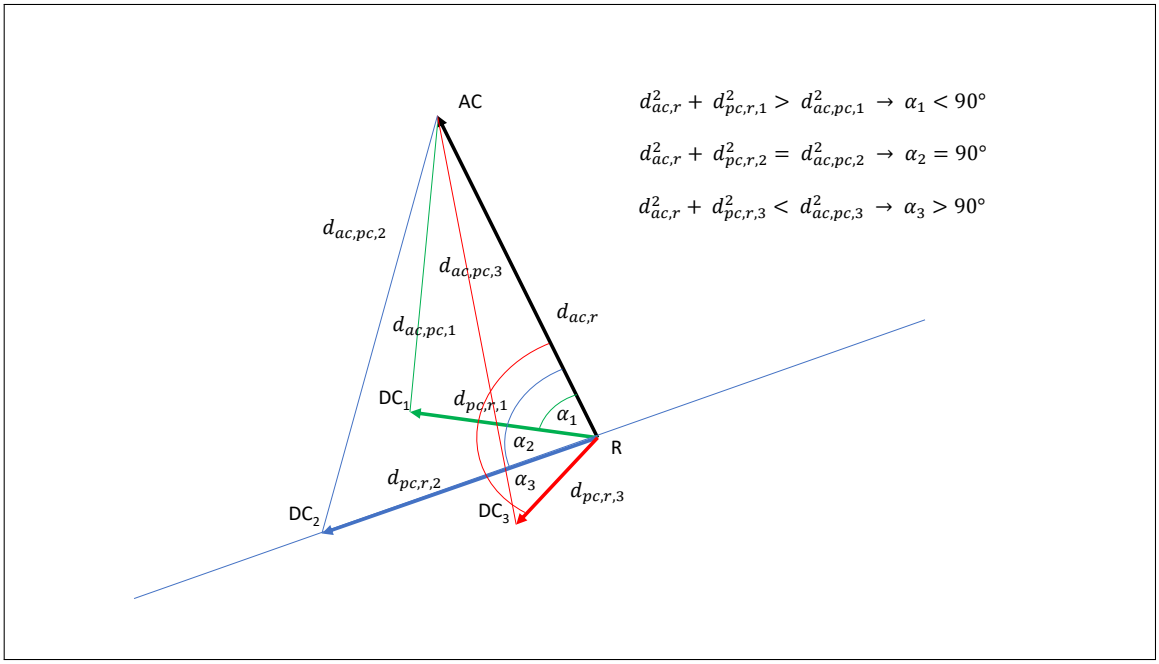


Figure 5.5: The candidates with an $\alpha < 90°$ count as valid candidates. A valid candidate can be seen in green, an invalid in red.

$$d^2_{ac,pc} < d^2_{ac,r} + d^2_{pc,r} \tag{5.1}$$

The valid candidates are placed in a list sorted by their distance from the appeared cluster. The candidates are tested in the order of their distance if the appeared cluster is really visible from them. This is necessary to avoid connecting a node to the tree through an

already explored wall. The appeared cluster is visible from the candidate, if following the vector pointing from the centroid of the candidate to the centroid of the cluster the value of the costmap cells does not exceed the threshold associated with an obstacle. The first candidate from which the appeared cluster is visible will be denoted as the best candidate and will be the parent of the new node.

If the cluster is not visible from any candidates it is registered as a fallen leaf. The list of fallen leaves serves as another blacklist.

When all appeared clusters are connected to a parent or registered as fallen leaves the old leaves of the previous iteration have to be managed. The cases are illustrated in Figure 5.6. The old leaves are marked with a blue square. If an old leaf is alive or it is not, but it has several new children there is nothing to be done. In the first case the old leaf (Figure 5.6b), in the second case its children (Figure 5.6a) are pushed to the list of new leaves. On the other hand, if an old leaf has no children and is not alive it means that the leaf is explored (Figure 5.6c). It is then marked as *explored* and those of its direct ancestors, which have no unexplored branches after the exploration of the leaf are also marked as *explored*. The leaf and its ancestors can be set *not explored* when in a later iteration the leaf is selected as the parent of an appeared cluster. For this to be possible the explored leaf is also pushed to the list of new leaves.

If an old leaf has only one child it means that the addition of the new cluster did not create a new branch (Figure 5.6d). In this case, there is no need to set up a new node, it is sufficient to replace the cluster of the old leaf with the newly appeared cluster. This way there could be a case when eventually the branch would cross a wall. Though this would not mean an error, for the sake of clear visibility and a better representation of the topology of the environment the clusters are replaced only if the new cluster is visible from the parent of the parent candidate. In any case, the node containing the new cluster will be pushed to the list of new leaves.

If the cluster is not connected to an old leaf but to its parent, then the parent is not changed other than receiving a new child and the appeared cluster is pushed to the list of new leaves.

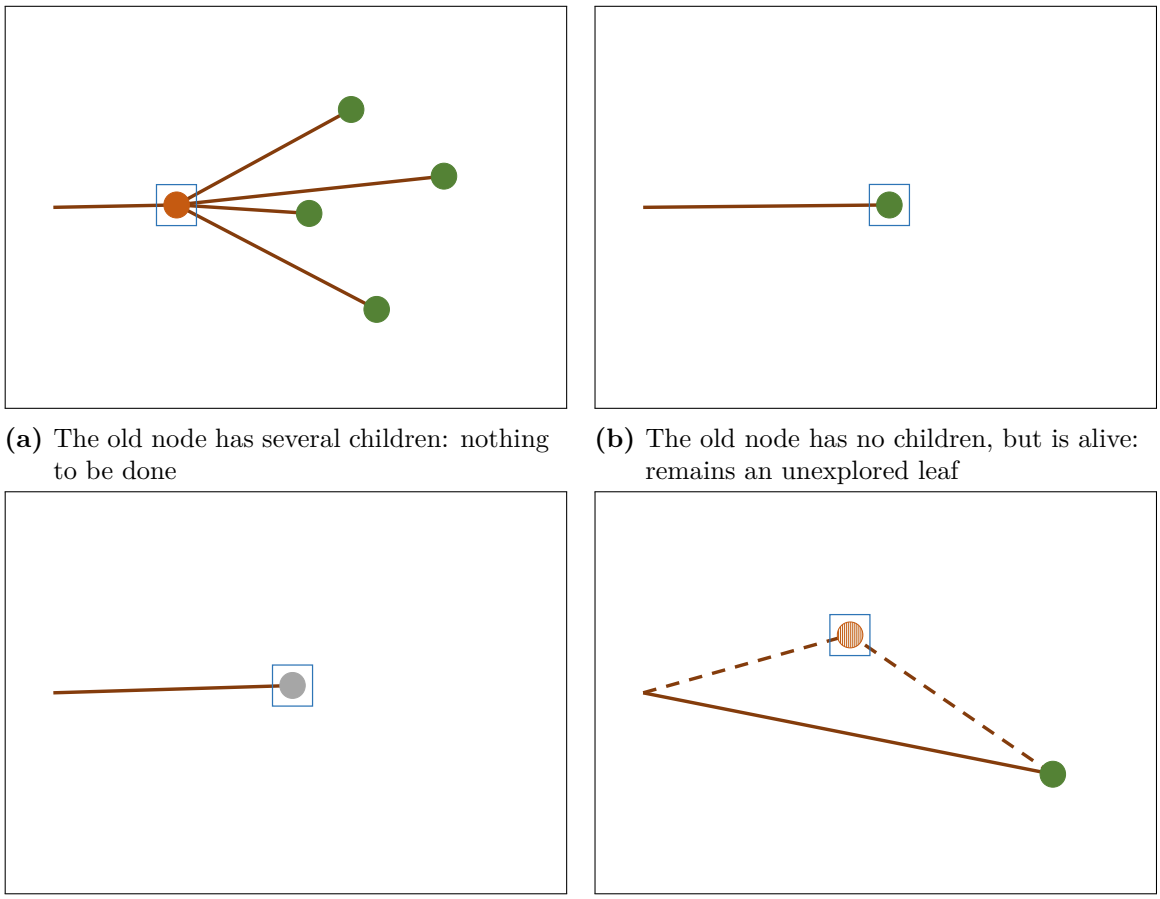
Finally, the new leaves are saved to serve as the old leaves of the next iteration and the building of the tree is finished. The cluster tree after a successful exploration can be seen in Figure 5.7.

5.4 Goal selection

5.4.1 Indices and costs of nodes

The goal selection implements a depth-first search on the tree built in the way described in the previous section. It is necessary for this search to be able to quickly identify if a node is a descendant of another given node. For this, the indices of the nodes will be introduced. It works as a prefix code following the layout of the tree.

The index of the root is an empty vector. Its children will have an index of one character, each their own index in the children vector of the root. Every child in the tree has the index of its parent completed with its own index in the parent's children vector. In this way, it can easily be determined if a node is a descendant of a parent node if the index of the descendant starts with the index of the parent. The indices of a tree can be seen in Figure 5.8, where the indices of each node are written in the circles representing the



(a) The old node has several children: nothing to be done

(b) The old node has no children, but is alive: remains an unexplored leaf

(c) The old node has no children and is not alive: it becomes explored

(d) The old node has one child: it gets replaced with it

Figure 5.6: The four scenarios of pruning. The old leaves are shown in blue squares, leaves are shown in green if they are alive and in grey if they are explored. Other unexplored nodes are shown in brown.

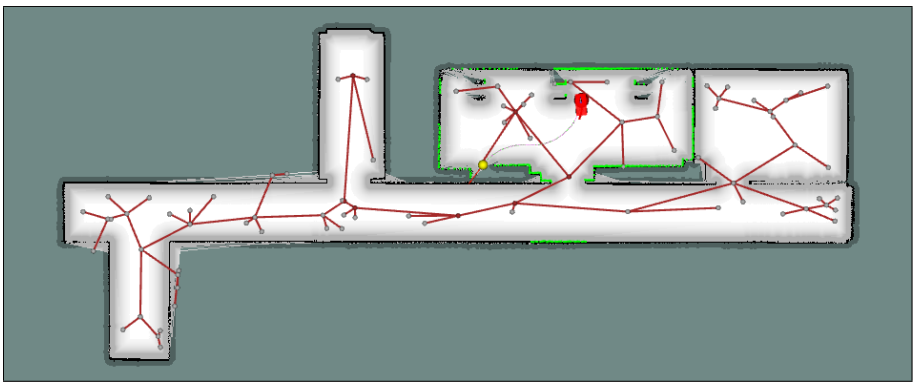


Figure 5.7: The cluster tree at the end of an exploration

nodes, while the nodes' indices in the children vector of their parents are indicated by the red numbers on the edges of the graph.

To be able to determine the closest unexplored branch when the pursued branch is explored the nodes need to have an associated cost with them. With the cost, the distances

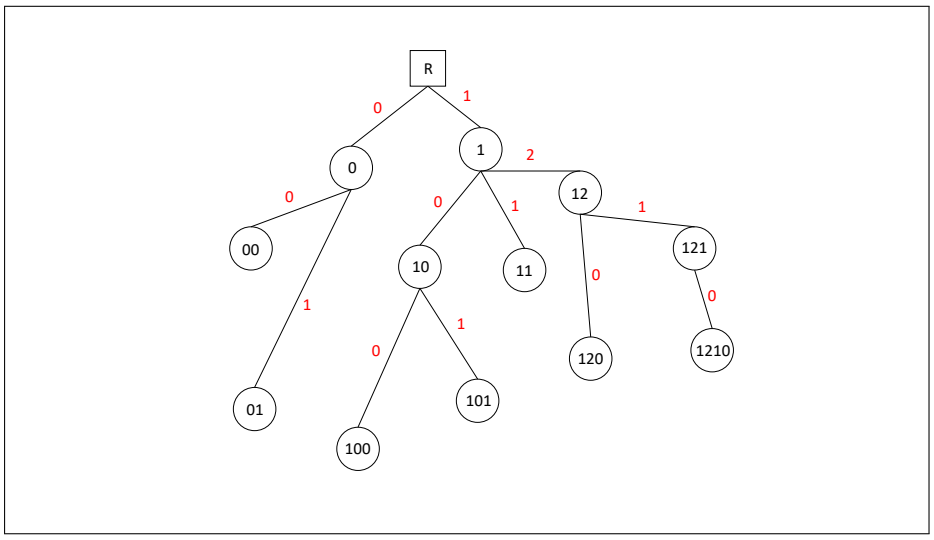


Figure 5.8: The indices of a tree structure

between nodes can be calculated not in euclidean space but travelling on the tree. This is important because the short euclidean distance between two nodes does not necessarily indicate proximity in the topology, for example when they are separated by a wall. We can introduce a cost on the tree with which we can easily calculate the travelling distance between nodes knowing the common ancestor.

Let the cost of the root be zero. We can also calculate the length of every segment of the tree using the coordinates of the centroid of the parent and the child. With these, a cost system can be built with the following method. The cost of every child will be the cost of its parent added to the length of the segment connecting them. The cost of a tree can be seen in Figure 5.9. In this case, the circles representing the nodes contain the cost of the given node, while the red numbers on the edges represent the euclidean distances between the parent and child node of the given edge.

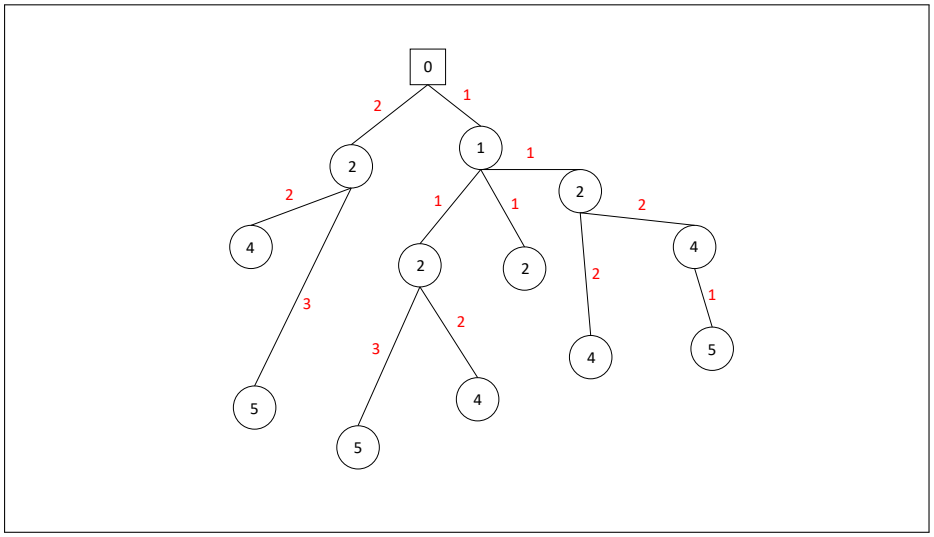


Figure 5.9: The cost of nodes in the tree structure

With this cost system, the distance travelling from any node (N_1) to another (N_2) can easily be calculated. First, we have to find their common ancestor (A) using the indices: the matching characters at the front of the index of each node will be the index of the

common ancestor. The travelling cost between the two nodes can be calculated as seen in (5.2), where $C(X)$ is the cost of X node and D is the distance travelled on the tree.

$$D = C(N_1) + C(N_2) - 2C(A) \tag{5.2}$$

5.4.2 The goal selection method

To select a new goal first we have to be familiar with the last goal of the exploration. Initially, this is the root of the tree.

First, we have to consider if the last goal is explored or not. If it is not, and it is a leaf of the tree it remains the goal of the exploration in the new iteration. If it is not explored, but it is not a leaf anymore, because the tree branched out from it we have to find a new goal. Since the branch is not entirely explored the goal has to be a descendant of the last goal to avoid abandoning unexplored areas midway and having to come back to them later as we have seen in the original frontier-based exploration. Therefore the new goal will be the closest unexplored leaf among the descendants of the last goal.

If the last goal is explored the first unexplored ancestor of the node has to be found. This can be done by stepping back through the parents until finding an unexplored node. The new goal can then be found as the closest unexplored leaf among the descendants of the ancestor.

Finding the closest descendant of a node is a problem that arises in two scenarios of goal selection. First, the descendants of the node have to be found. This is achieved by comparing the indices of the leaves of the tree with the index of the ancestor and selecting those from them which contain it at their beginning. After this, the cost of travelling from the last goal to the selected leaves is calculated with (5.2) and the one with the lowest cost will be selected as the next goal of exploration. The goal selection and tree-building process are demonstrated in Figure 5.10.

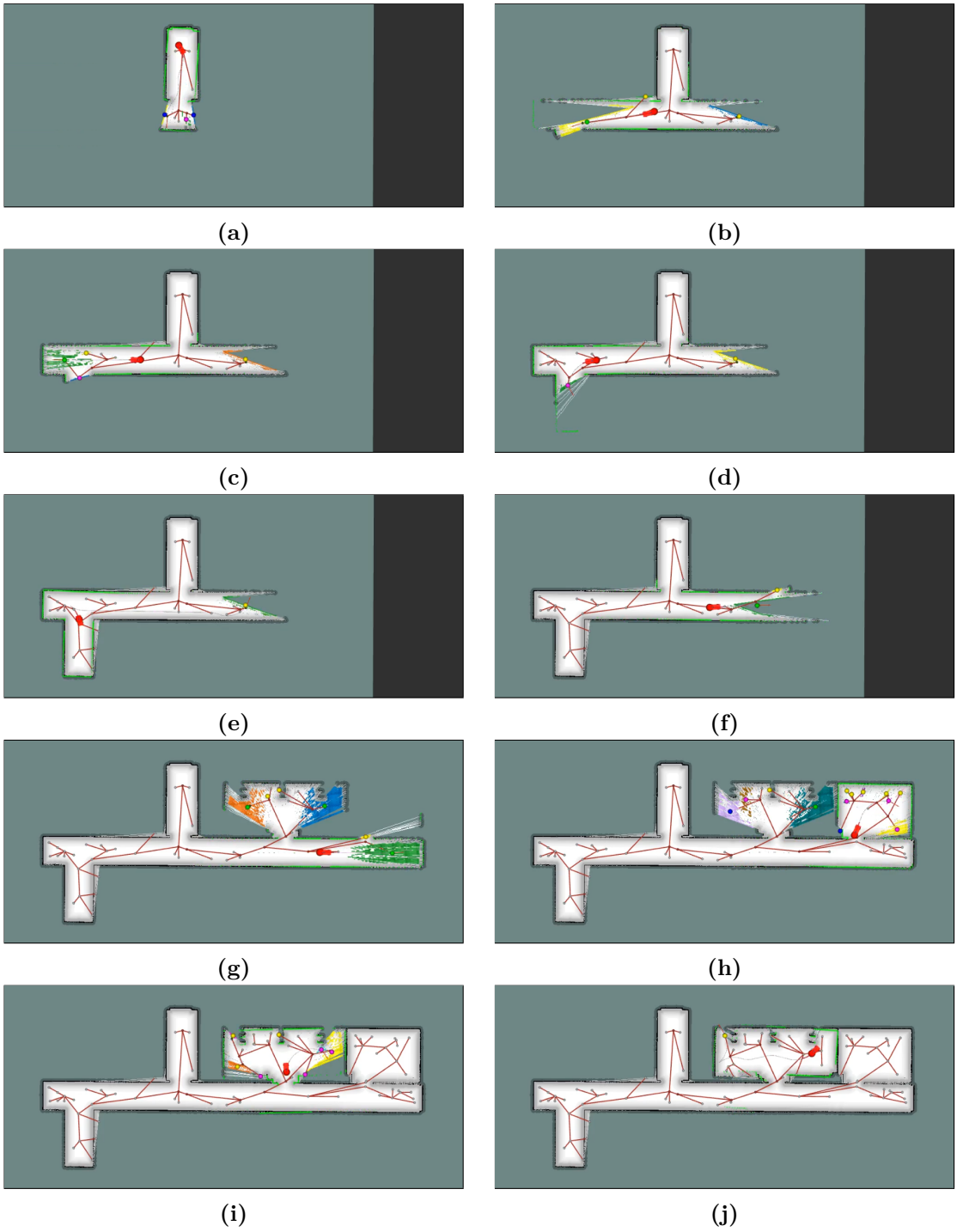


Figure 5.10: Steps of a cluster_tree exploration

Chapter 6

Experimental results

To validate the claims of the previous chapter it is necessary to confirm the improvements in the exploration method using various test environments. In our case, the tests were run in a simulated environment. It will be described in more detail in the following section.

6.1 Test environment

The robot is simulated using Gazebo along with all its sensors. The physical characteristics and parameters simulated are consistent with the real-life robot. All program components not responsible for the simulation itself operate the same way as they would when using the real-life hardware and are in no way notified of the nature of the environment.

There are several worlds modelled in Gazebo but the one used for the comparison of the discussed methods is modelled after two neighbouring rooms and the connecting section of a corridor, which is part of the ground floor of a building of the university. The Gazebo world is shown in Figure 6.1.

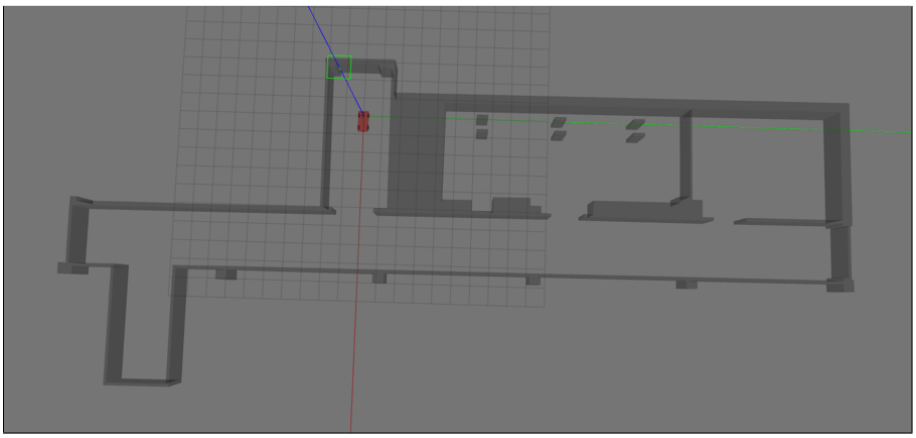


Figure 6.1: The test world shown in Gazebo

The measured values to be compared are the following. The time elapsed during the exploration, the progress of completeness and the distance travelled. The first indicator is simply calculated as the difference between the finishing and starting point in time.

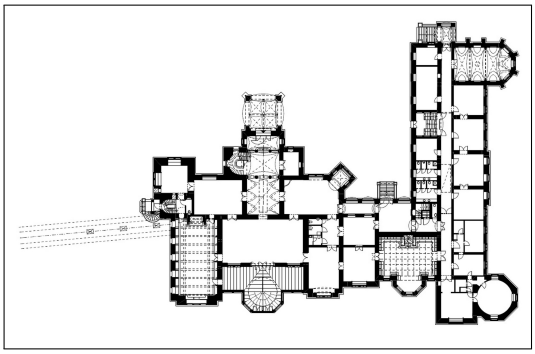
The progress of the exploration is the percentage of the map that is already explored at any given moment. It is registered as a function of time. To calculate the momentary

value of progress, a finished map is captured and stored by the map_saver service of the map_server ROS module. The current map is evaluated with respect to the finished map in every iteration. The value of progress at a moment equals the quotient of the number of cells which have the same value in both maps and the number of all cells in the finished map.

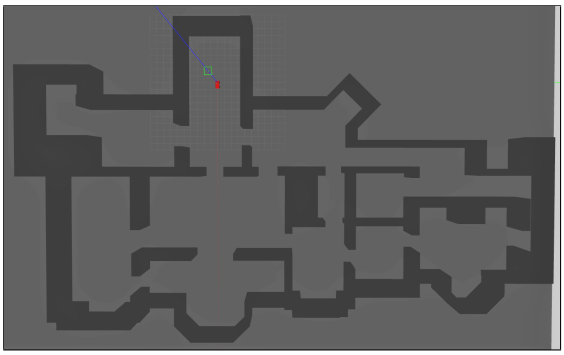
Thirdly the distance travelled is calculated by integrating the position of the robot in every iteration.

Both methods of exploration were tested in simulation for a total of ten times in this world.

Another test environment is modelled after the Nádasdy Mansion located in Nádasdladány. The ground plan can be found in [22] and is shown in Figure 6.2a. The plan was reduced to the area currently open to the public and further simplified for modelling. The resulting Gazebo world can be seen in Figure 6.2b. This is a significantly bigger and more complex world for the algorithms to be tested on. There were five test cases documented for each of the methods exploring this environment.



(a) The ground plan of the Nádasdy Mansion [22]



(b) The simplified Nádasdy Mansion modelled in Gazebo

Figure 6.2: The layout of the Nádasdy Mansion in the real life and in the Gazebo world

6.2 Simulation results

6.2.1 Classical frontier-based exploration

First, let us look at the simulation results of the original frontier-based method improved with the ideal angle of approach. The progress of the exploration can be seen in Figure 6.3. Every fine line represents a test while the thick red line shows the average progress at any given moment. The measured values can be found in Table 6.1. The average time to get to 90% progress of exploration was 259.5 seconds, while the average time for finishing the simulation (when no reachable frontiers are left) was 528 seconds. The average distance travelled accounted to 138.74 m.

There are cases when the global planner takes an unreasonably long time to turn around the robot in the narrow corridor instead of in a doorway. This results in plateaus in the progress curve, which are more noticeable in Figure 6.6 of the cluster tree exploration. In other cases, the issue does not occur. To make the results more independent of the

planners the worst three test cases were not taken into account in the comparison of the results. The upgraded average curve can be seen in Figure 6.3 with a blue dashed line.

Progress time [s]	90%	221	246	301	256	216	
	finish	446	446	316	626	711	
Distance [m]		135.08	142.81	97.43	180.49	130.96	
							average
Progress time [s]	90%	231	246	271	286	321	259.5
	finish	466	531	416	461	861	528.0
Distance [m]		146.88	115.85	128.15	146.68	163.08	138.74

Table 6.1: Experimental results of the explore_lite method exploring the corridor

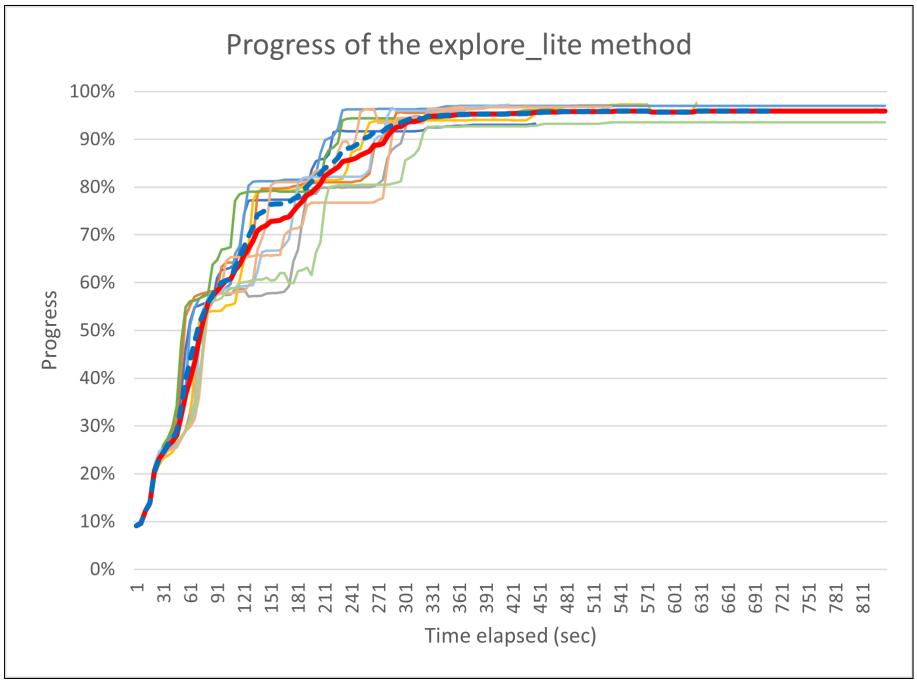


Figure 6.3: The progress over time using the improved explore_lite to explore the corridor. The average of the progress is shown in red, the updated average is represented by the blue dashed line.

The first thing to be mentioned is the importance of taking into consideration the constraints forced by the physical construction of the robot. Since the original explore_lite package was made for a holonomic robot the angle of approach was a constant vector in the direction of the x-axis. During tests run with the original package, the car-like robot would have to approach every goal on the corridor perpendicular to the walls, which led to it being stuck the majority of the time, never finishing the exploration.

On the other hand, the improved method was able to finish the exploration most of the time and provided more natural trajectories for the non-holonomic robot.

Though relevantly better than the original, even the improved method could not eliminate the problems caused by greedy goal selection. The biggest pitfall of it is leaving rooms partly unexplored (visible in Figure 6.4), and after visiting every room, which could have meant the end of exploration, it has to go back to finish exploring each room wasting a lot of time in the process.

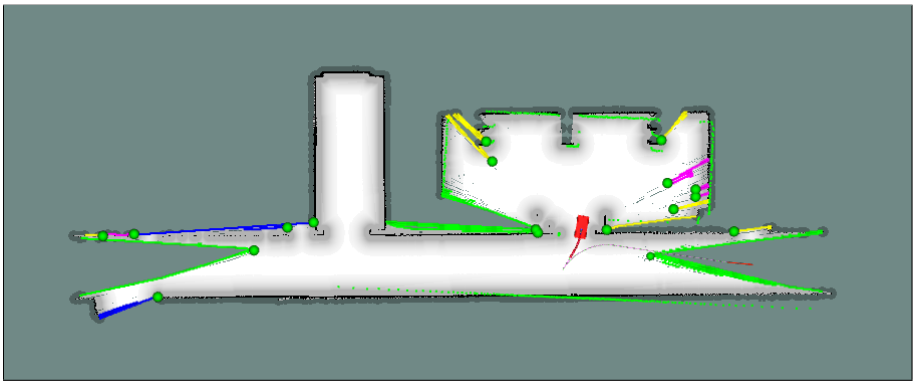


Figure 6.4: The robot leaves the room before finishing its exploration

A smaller scale oscillation can also be observed when the greedy goal selection is torn between two goal candidates and the manoeuvring of the robot is enough to change the direction of the next goal, leaving the robot going back and forth in one place.

The results of exploring the Nádasdy Mansion with the explore_lite method can be seen in Table 6.2. The times and distances are higher than in the case of the corridor, simply because this world is much bigger and more complex. Observing the values we can see that increasing the complexity of the exploration made the problem of visiting the same places more times less relevant than in the corridor. This happened because the looped structure of the map already made it necessary to go through the same areas more than once. So while in the case of the corridor the time spent after reaching 90% progress was more than half of the total time of the exploration, in this case, that value is only 36%. The progress over time can be seen in Figure 6.5. In this case, the erasure of some of the results was not necessary, because the corridors were wide enough for the robot to turn back on and there were more doorways, so the turning did not take an unreasonably long time like in the case of the corridor.

							average
Progress time [s]	90%	1066	796	1026	676	741	861.00
	finish	1838	1076	1361	1011	1491	1355.40
Distance [m]		503.44	361.58	451.40	332.13	499.03	429.51

Table 6.2: Experimental results of the explore_lite method exploring the Nádasdy Mansion

6.2.2 Tree-based method for exploration

The tree-based method provided a reliable way of exploration producing an average of 303 s travelling time among the ten simulations, and a slightly lower value of 280 s to get to 90% progress. The progress during the explorations is shown in Figure 6.6. The colouring is the same as we have seen in the case of explore_lite. The effectiveness of the method can be observed in the distance travelled, which averaged a value of 93.12 m.

This can be traced back to the lack of unintentional backtracking during the exploration. Following the tree structure, the robot is not allowed to leave a room until it is completely explored so when the robot does leave the room there is no need to come back to it later. The process of exploration following the tree can be seen in the set of Figures 5.10.

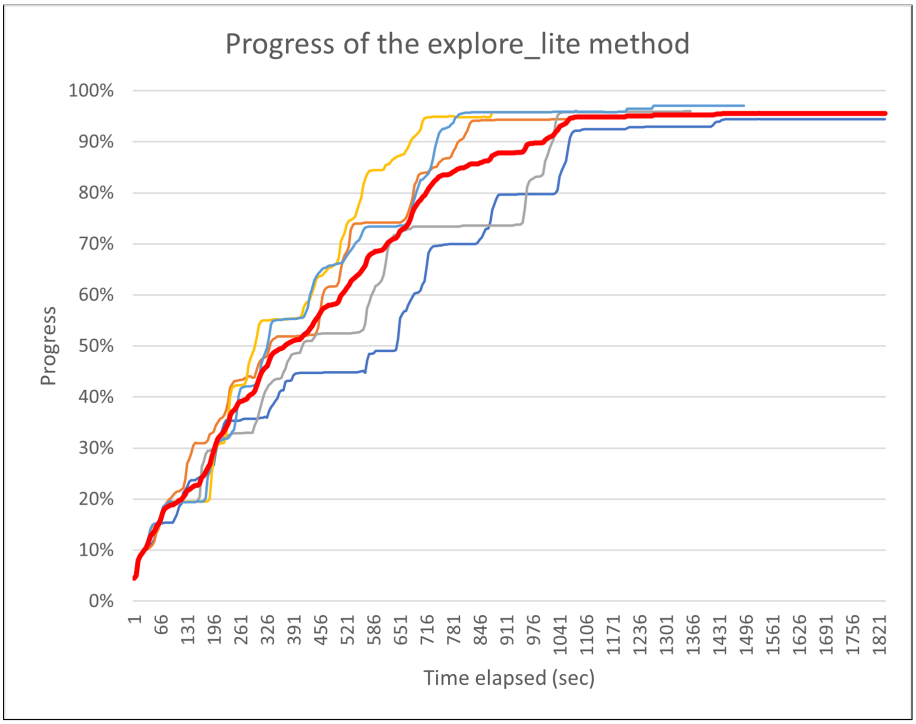


Figure 6.5: The progress over time using the improved explore_lite to explore the Nádasdy Mansion. The average of the progress is shown in red.

Progress time [s]	90%	246	216	276	316	211	
	finish	256	226	301	321	231	
Distance [m]		85.79	77.54	101.84	88.17	79.44	
							average
Progress time [s]	90%	261	276	286	336	401	282.5
	finish	276	276	316	346	481	303.0
Distance [m]		91.35	93.10	99.86	116.51	97.56	93.12

Table 6.3: Experimental results of the cluster_tree method exploring the corridor

We can see on the diagram, that the curves of different tests proceed together for a long time. Then individual strands get left behind. Watching the simulation in real-time explains to this phenomenon. The car takes a long time to proceed with the exploration when the global planner tries to turn it around in the narrow corridor. In the quickest explorations, the car made a Y-turn in a doorway instead. To make this issue less relevant in the result the worst three test cases were deleted from the comparison, similarly to the case of explore_lite. The updated average curve is shown in a dashed blue line in Figure 6.6.

The results of exploring the Nádasdy Mansion with the cluster_tree method can be seen in Table 6.4. As the method relies on building a tree for goal selection it did not perform as well in the looped environment of the mansion as in the already tree-structured corridor. Often the closest unexplored branch in the tree was not the closest in the environment causing the robot to travel further away than it would have needed to. It is also due to the looped structure that sometimes rooms on a loop remain unexplored if the robot follows the loop first, similarly to how the greedy algorithm leaves rooms half-explored

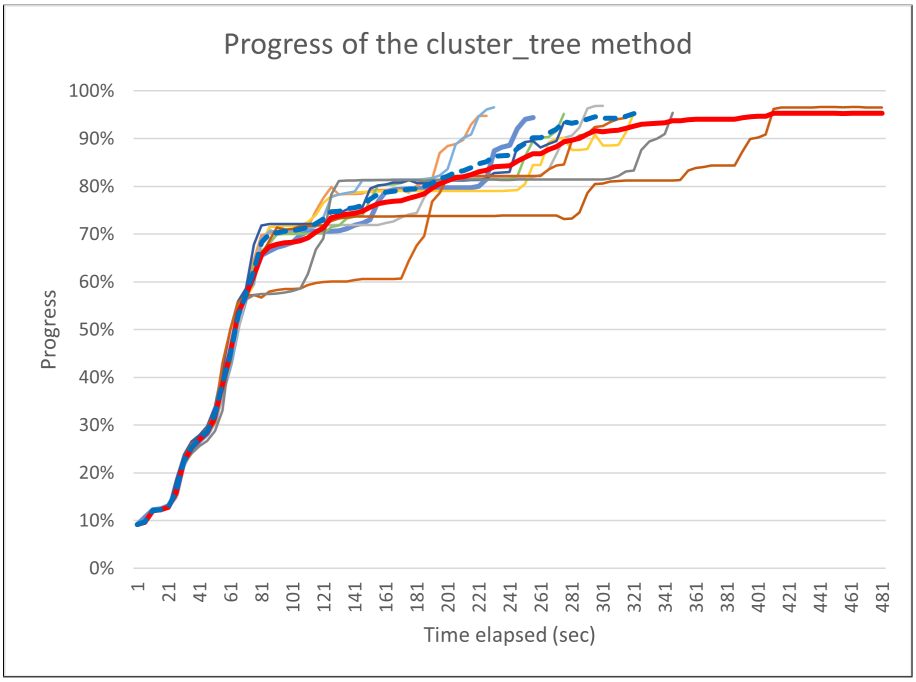


Figure 6.6: The progress over time using the cluster_tree method to explore the corridor. The average of the progress is shown in red, the updated average is represented by the blue dashed line.

and in both cases, the robot has to come back to the room later. This is the cause for the plateaus at the end of the curves of each test case in Figure 6.7.

The steps of the exploration are shown in the set of Figures 6.8. Though the exploration was successful, the bigger map and its consequences also took their toll on the speed of the exploration. Due to the high number of leaf nodes to be examined, the building of the tree proved to be highly compute-intensive. As a result the building of the tree often halted for several seconds after discovering new clusters. This, however, did not introduce any errors in the exploration, as the robot pursued the last goal nonetheless, and when it reached its destination, waited until a new goal could be assigned to it. So while it can be said, that the high computational times did not interfere with the success of the exploration, they did cause a delay in progress. The exploration lasted an average of 1122 seconds. This means a total of 18 minutes and 42 seconds.

							average
Progress	90%	1026	1016	1041	851	936	974.00
time [s]	finish	1076	1066	1286	1146	1036	1122.00
Distance [m]		360.68	371.46	444.68	381.69	315.64	374.83

Table 6.4: Experimental results of the cluster_tree method exploring the Nádasdy Mansion

6.2.3 Comparison of the test results

To compare the improved explore_lite and the tree-based method we can look at their progress over time side by side in Figure 6.9. The shown curves are the updated average

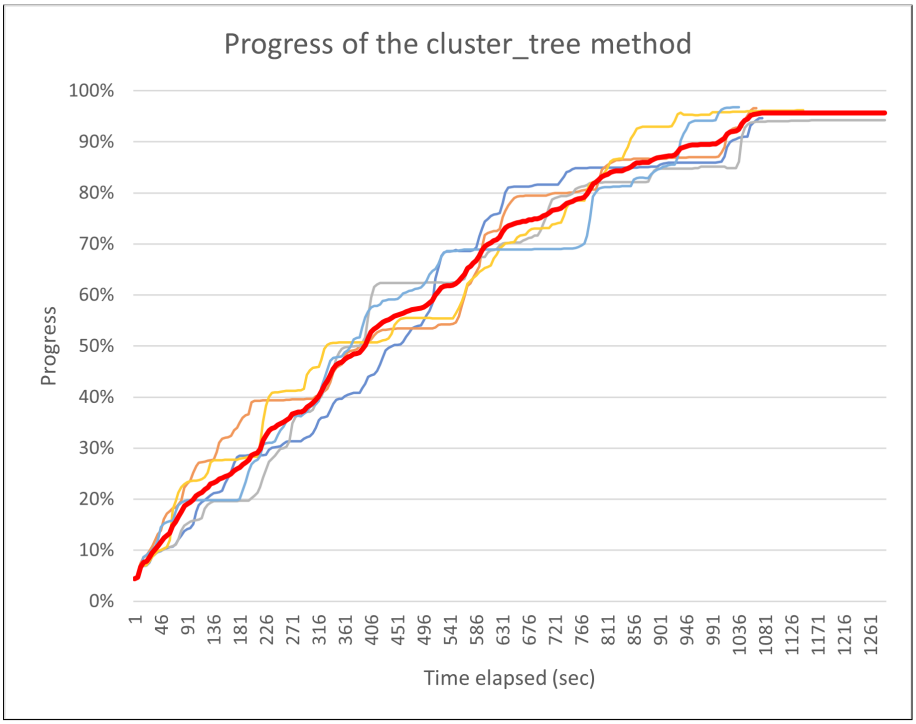


Figure 6.7: The progress over time using the cluster_tree method to explore the Nádasdy Mansion. The average of the progress is shown in red.

curves of the two exploration methods, where the worst three test cases were deleted due to issues with the global planner. The average progress of explore_lite is marked in red and the average progress of the cluster tree method is shown in blue.

One thing we can notice is that the two methods do not seem to be much different in the first part of the exploration, though the green curve comes before the red one from 40% to 75%, but they reach 90% almost at the same time. The explore_lite achieves this in an average of 245 seconds, while the cluster_tree in 257.43 seconds. If we look at differences in the time elapsed in individual test cases we can say that this is not a very significant difference.

The real difference between the two methods though is how that 90% of progress is distributed. Using the cluster_tree method the map consists of unknown and completely known areas. On the other hand, the explore_lite first takes a general look at every place, thus gathering 90% of the information slightly more rapidly, but with unknown areas all over the map. This has to be corrected later, which leads to the slow and movement-intensive collection of the remaining 10% of the information. This can be seen in the logarithmic-like nature of the curves in Figure 6.3 and the average curve in Figure 6.9, continuing long after achieving a high percentage relatively quickly. This makes the average time from the 90% mark to the finishing of the exploration 260.71 seconds, which takes a longer time than reaching 90% of progress from the beginning.

On the other hand, the exploration with the cluster_tree method ends shortly after getting to the 90% mark. Since the known part of the map consists of already completely explored areas, thanks to the systematic nature of the exploration, there is no need to return to previously visited territories. Thanks to this it takes only an average of 12.14 seconds to finish the exploration after reaching the 90% mark.

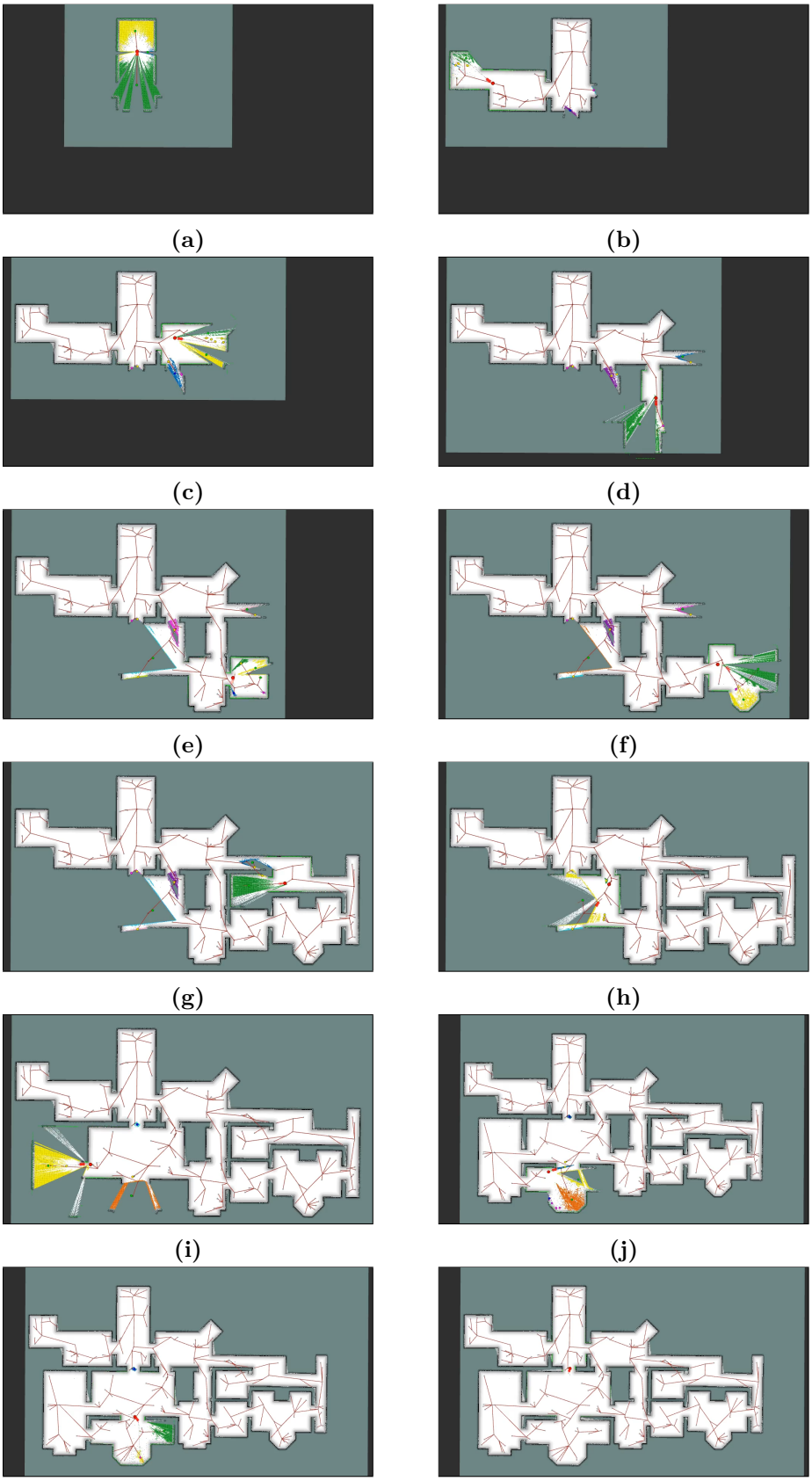


Figure 6.8: Steps of the exploration of the Nádasdy Mansion

This results in the average total travel time being reduced to its 53% (from 506.0 s to 269.57 s) using the cluster_tree method. The difference is also evident if we look at the differences in the distance travelled. Using the tree-based method the average distance toured by the robot was reduced from 134.64 m to 88.18 m, its 65.5%, which means that about the third of the distance travelled by the robot using the explore_lite method was unnecessary and due to the greedy goal selection algorithm.

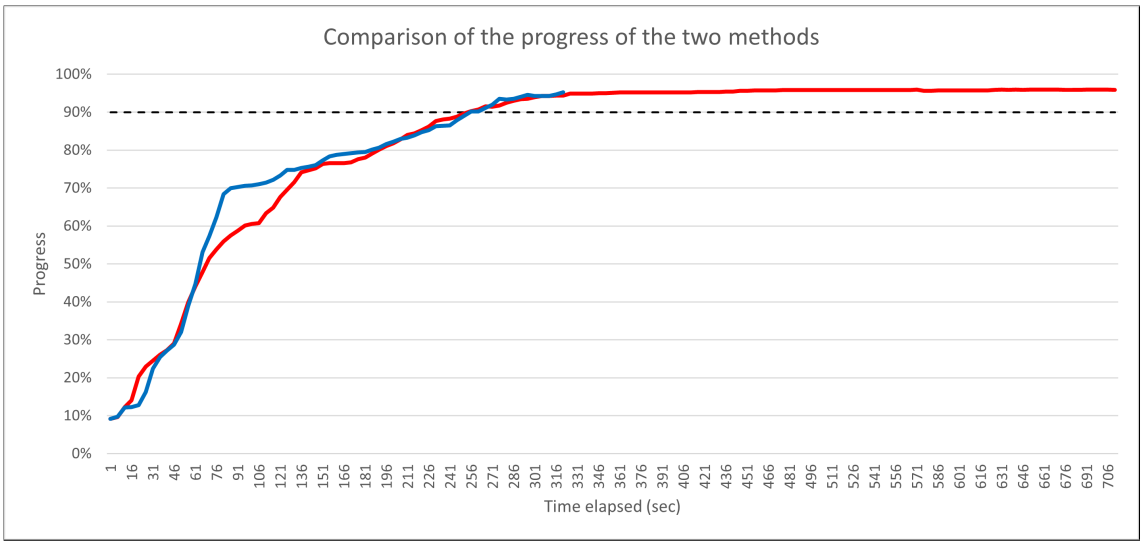


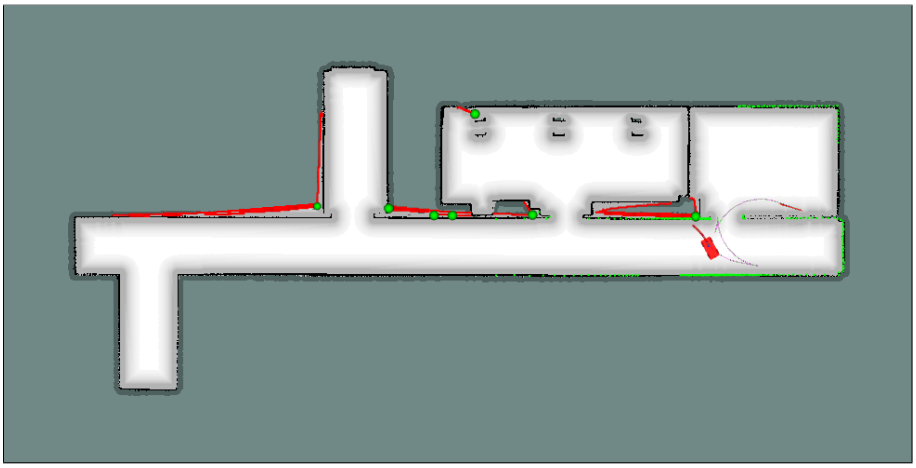
Figure 6.9: Comparison of progress using the two methods to explore the corridor. The updated average of the explore_lite is shown in red while the updated average of the cluster_tree is shown in blue.

Another aspect of the performance of exploration is the quality of the built map. The maps generated by the two methods are generally very similar, though maps generated exploring with explore_lite tend to be slightly more detailed. This can be seen in Figure 6.10, especially in the bigger room. This can be the result of two factors.

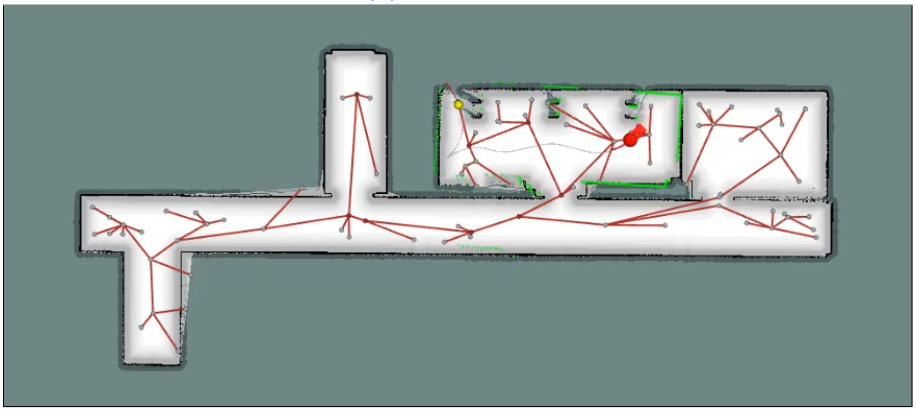
As we have already seen, the explore_lite takes more time exploring, often visiting the same areas multiple times, allowing it to take repeated measurements in the same place, often from different angles.

The other contributing factor could be the number of frontiers. In the tree-based method, the frontier points are clustered, leading to a smaller number, but bigger frontiers. On the other hand, explore_lite uses its frontiers without any after-processing which leads to a higher number of smaller frontiers. Then, as all frontiers can be goals of the exploration regardless of their size, a more detailed map is formed, but the exploration takes a longer time.

In the case of the Nádasdy Mansion, the looped structure of the map made the problems of explore_lite's greedy goal selection less relevant and the tree structure of the cluster_tree method less effective, which resulted in the decreasing of the difference between the results of the two explorations. There were test cases when explore_lite performed better, though looking at the average performance of the two methods the cluster_tree method still shows better results. While in the corridor following the tree the robot travelled only 53% of the total time of using greedy goal selection, in the case of the Mansion this value rose to 83%. Similarly, the proportion of distance travelled using the two methods rose from 65% to 87%. The comparison of the methods' progress over time is shown in Figure 6.11. We can



(a) explore_lite



(b) cluster_tree

Figure 6.10: Final maps build by the two methods of exploration

see that the curves rise almost in the same way and plateau at the same point. Despite this, the exploration using the cluster_tree method finishes an average of 233.4 seconds sooner than in the case of explore_lite.

6.3 Summary of the results

In this chapter, we have seen both exploration methods in action. The two Gazebo worlds used for testing showed different aspects of the methods. The tree-like structure of the corridor allowed the cluster_tree method to work to its full capacity, reducing the travelling time to 53% and the distance travelled to 65% in comparison with explore_lite.

On the other hand, the Nádasdy Mansion had a looped structure, which made the exploration following a tree structure less effective and visiting some places more than once unavoidable. The latter made the greedy goal selection of the explore_lite a less wasteful strategy while simultaneously making the following of a tree structure more expensive. This is reflected in the decrease in the difference between the two methods. In this case, the travelling time was only reduced to 83% and the distance travelled to 87%.

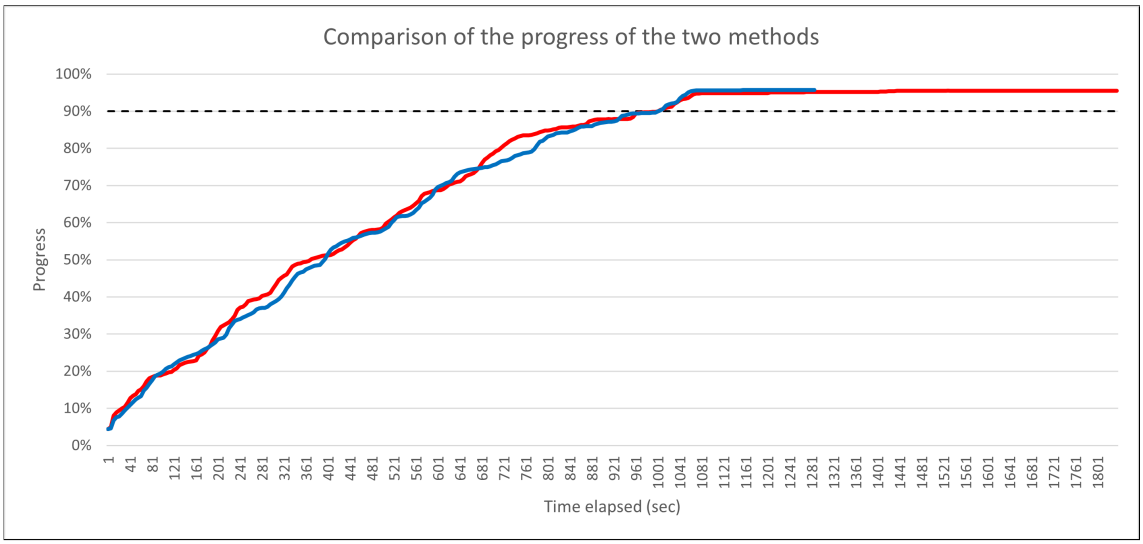


Figure 6.11: Comparison of progress using the two methods to explore the Nádasdy Mansion. The average of the explore_lite is shown in red while the average of the cluster_tree is shown in blue.

Chapter 7

Conclusion

In this paper we have reviewed a frontier-based method of autonomous exploration and modified it to be usable on a robot limited by non-holonomic constraints, then proposed a new, tree-based approach to goal selection to eliminate the problems seen with the greedy algorithm. The simulation results showed a significant improvement in the time and distance travelled of the exploration when the map also followed a tree-like structure and a slight improvement in the case of a looped map.

The latter is caused by the tree structure not being able to register different branches connecting with each other. In this way, the tree gives a false representation of the structure of the environment and it is thus an erroneous way to think that the closest unexplored branch could be determined using nothing else, but the tree itself.

The solution to this could be to register the proximity of neighbouring branches using shortcuts where the tree would close to a loop and determine the best goal from all the closing branches and ultimately choose the closest of these nodes. The distance of the nodes can in this case defined as their distance following the tree, as they are calculated from their corresponding branches. This can be a promising direction for further improving the algorithm.

The other problem of exploration with the tree structure in a large environment was the high computation times of finding the best node to connect the newly found frontiers to. This could be reduced by only considering candidates in the given proximity of the new cluster and using a different method to sort the remaining parent candidates. With this, a faster exploration would be possible even when the tree has a high number of nodes.

In conclusion, the proposed algorithm proved to be a great improvement on the original method in environments following a tree structure, like most offices, and performed better even in an environment not favourable for tree-like exploration, still bringing forward promising ways for further development of the algorithm.

Bibliography

[1] Márton Antal. Navigation algorithms for car-like robots, 2021.

[2] Bence Balogh. Pályakövetési algoritmusok fejlesztése intelligens járműplatformra, 2020.

[3] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Practical search techniques in path planning for autonomous driving. *Ann Arbor*, 1001(48105): 18–80, 2008.

[4] Charles DuHadway. explore Package Summary (19th of October, 2022). `http://wiki.ros.org/explore`.

[5] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part I. *IEEE Robotics & Automation Magazine*, 13(2):99–110, 2006. DOI: `10.1109/MRA.2006.1638022`.

[6] Haitham El-Hussieny, Samy F. M. Assal, and Mohamed Abdellatif. Improved Backtracking Algorithm for Efficient Sensor-Based Random Tree Exploration. In *2013 Fifth International Conference on Computational Intelligence, Communication Systems and Networks*, pages 19–24, 2013. DOI: `10.1109/CICSYN.2013.17`.

[7] Eleobert. Density-based spatial clustering of applications with noise (DBSCAN) (1st of September, 2022). `https://github.com/Eleobert/dbscan`.

[8] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[9] Jan Faigl and Miroslav Kulich. On determination of goal candidates in frontier-based multi-robot exploration. In *2013 European Conference on Mobile Robots*, pages 210–215, 2013. DOI: `10.1109/ECMR.2013.6698844`.

[10] Antonio Franchi, Luigi Freda, Giuseppe Oriolo, and Marilena Vendittelli. The sensor-based random graph method for cooperative robot exploration. *IEEE/ASME Transactions on Mechatronics*, 14(2):163–175, 2009. DOI: `10.1109/TMECH.2009.2013617`.

[11] Brian Gerkey. gmapping Package Summary (31th of October, 2022). `http://wiki.ros.org/gmapping`.

[12] David E Goldberg. *Genetic algorithms*. pearson education India, 2013.

[13] Jiří Hörner. explore_lite Package Summary (19th of October, 2022). `http://wiki.ros.org/explore_lite`, .

[14] Jiří Hörner. m-explore GitHub Repository (19th of October, 2022). https://github.com/hrnr/m-explore, .

[15] Jiří Hörner. Map-merging for multi-robot system, 2016. URL https://is.cuni.cz/webapps/zzp/detail/174125/.

[16] Jinho Kim, Kie Jeong Seong, and H. Jin Kim. An efficient backtracking strategy for frontier method in Sensor-based Random Tree. In *2012 12th International Conference on Control, Automation and Systems*, pages 970–974, 2012.

[17] Jong Min Kim, Kyung Il Lim, and Jung Ha Kim. Auto parking path planning system using modified Reeds-Shepp curve algorithm. In *2014 11th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pages 311–315, 2014. DOI: 10.1109/URAI.2014.7057441.

[18] Jie Liu, Yong Lv, Yuan Yuan, Wenzheng Chi, Guodong Chen, and Lining Sun. A Prior Information Heuristic based Robot Exploration Method in Indoor Environment. In *2021 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pages 129–134, 2021. DOI: 10.1109/RCAR52367.2021.9517416.

[19] Eitan Marder-Eppstein. move_base Package Summary (19th of October, 2022). http://wiki.ros.org/move_base.

[20] G. Oriolo, M. Vendittelli, L. Freda, and G. Troso. The SRT method: randomized strategies for exploration. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, volume 5, pages 4688–4694 Vol.5, 2004. DOI: 10.1109/ROBOT.2004.1302457.

[21] Jhielson M. Pimentel, Mário S. Alvim, Mario F. M. Campos, and Douglas G. Macharet. Information-Driven Rapidly-Exploring Random Tree for Efficient Environment Exploration. *Journal of Intelligent & Robotic Systems*, 91(2):313–331, Aug 2018. ISSN 1573-0409. DOI: 10.1007/s10846-017-0709-0. URL https://doi.org/10.1007/s10846-017-0709-0.

[22] Viola Pleskovics. Visszaforog az idő kereke? – A nádasdladányi Nádasdy-kastély újjáteremtési programja (1st of October, 2022). https://shorturl.at/hnpSU.

[23] Roland Putnoki. Autószerű robotok pályatervező algoritmusának fejlesztése rácsalapú térképben, 2018.

[24] Gábor Péceli. Measurement Theory Lecture Notes, 2019.

[25] ROS. ROS.org (31th of October, 2022). https://www.ros.org/.

[26] Yujie Tang, Jun Cai, Meng Chen, Xuejiao Yan, and Yangmin Xie. An autonomous exploration algorithm using environment-robot interacted traversability analysis. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4885–4890, 2019. DOI: 10.1109/IROS40897.2019.8967940.

[27] Hassan Umari and Shayok Mukhopadhyay. Autonomous robotic exploration based on multiple rapidly-exploring randomized trees. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1396–1402, 2017. DOI: 10.1109/IROS.2017.8202319.

[28] B. Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'*, pages 146–151, 1997. DOI: 10.1109/CIRA.1997.613851.