MULTI-ROBOT MAP EXPLORATION BASED ON MULTIPLE RAPIDLY-EXPLORING RANDOMIZED TREES

by

Hassan Abdul-Rahman Umari

A Thesis Presented to the Faculty of the
American University of Sharjah
College of Engineering
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in
Mechatronics Engineering

Sharjah, United Arab Emirates

May, 2017

© 2017 Hassan Abdul-Rahman Umari. All rights reserved.

Approval Signatures

We, the undersigned, approve the Master's Thesis of Hassan Abdul-Rahman Umari.

Thesis Title: Multi-Robot Map Exploration Based on Multiple Rapidly-Exploring Randomized Trees

Signature	**Date of Signature** (dd/mm/yyyy)
___________________________ Dr. Shayok Mukhopadhyay Assistant Professor, Department of Electrical Engineering Thesis Advisor	_______________
___________________________ Dr. Hasan Mir Associate Professor, Department of Electrical Engineering Thesis Committee Member	_______________
___________________________ Dr. Mamoun Abdel-Hafez Professor, Department of Mechanical Engineering Thesis Committee Member	_______________
___________________________ Dr. Lotfi Romdhane Director, Mechatronics Engineering Graduate Program	_______________
___________________________ Dr. Mohamed El-Tarhuni Associate Dean for Graduate Affairs and Research, College of Engineering	_______________
___________________________ Dr. Richard Schoephoerster Dean, College of Engineering	_______________
___________________________ Dr. Khaled Assaleh Interim Vice Provost for Research and Graduate Studies	_______________

Acknowledgements

First of all, I would like to thank God, the most gracious and the most merciful, for all his blessings, gifts, and life lessons. I would also like to thank my beloved parents and my family, without whose help and support I wouldn't have been able to reach this far.

I would also like to extend my gratitude to my advisor Dr. Shayok Mukhopadhyay, for his guidance, patience and help. My thanks also go to the Mechatronics Engineering program for giving me the opportunity to do my Master's here at AUS. Special thanks to all faculty members who taught me during my time here.

I would also thank all my friends with whom I spent unforgettable times: Wasim Al-Masri, Ehab I. Al Khatib, Ali Qahtan, Mohamed Elmustafa, Abdul-Rahman Renawi and all the others.

I would also like to thank Mr. Kent Roferos for his help in the MTR lab, and Mr. Wasil El Tahir for his great help in allowing me to use three Kobuki robots, without which I wouldn't have been able to do my experiments.

Finally, I would like to thank my committee members, Dr. Mamoun Abdel-Hafez and Dr. Hassan Mir, for the effort and time they dedicated to reading my proposal and thesis reports, and for their valuable feedback.

Dedicated to my mother Lebana, and my father Abdul-Rahman . . .

Abstract

Efficient robotic navigation requires a predefined map. In order to autonomously acquire a map, it is desired that robots have the ability to explore unknown environments with minimum cost and time, while ensuring complete map coverage. Meeting these requirements is challenging, and has attracted a lot of research. Various autonomous map exploration strategies exist, which direct robots to unexplored space by detecting frontiers. Frontiers are boundaries separating known space form unknown space. Usually frontier detection utilizes image processing tools like edge detection, thus limiting it to two dimensional (2-D) exploration. In this work we present a new exploration strategy based on the use of multiple Rapidly-exploring Random Trees (RRTs). The RRT algorithm is chosen because it is biased towards unexplored regions. Also, using RRT provides a general approach which can be extended to higher dimensional spaces. The proposed strategy is implemented and tested using the Robot Operating System (ROS) framework. Additionally this work uses local and global trees for detecting frontier points, which enables efficient robotic exploration. Further more, a market-based task allocation strategy for coordination between multiple robots is adopted. Simulations and experimental results show that the proposed strategy can successfully extract frontiers, and explore the entire map in a reasonable amount of time, and with a reduced map exploration cost. It is also shown in this work that the proposed approach has the above mentioned performance benefits without substantially losing performance when compared against image processing-based frontier detection techniques in two dimensional spaces.

Search Terms: ***Mapping, Exploration, ROS, Multi Agent, RRT***

Table of Contents

Abstract 6

List of Figures 10

1. Introduction 12

    1.1 Literature Review 14

    1.2 Motivation 17

    1.3 Thesis Organization 18

2. Background 19

    2.1 Path Planning 19

        2.1.1 Common terminologies used in RRT and Dijkstra's algorithm . 19

        2.1.2 The RRT algorithm 20

        2.1.3 Dijkstra's algorithm 21

    2.2 ROS Background 21

        2.2.1 ROS concepts 24

        2.2.2 ROS coordinate frames 25

        2.2.3 ROS launch files 27

    2.3 Map Representations 27

        2.3.1 Volumetric maps 28

        2.3.2 Feature-based maps 29

        2.3.3 Topological maps 29

    2.4 Frontier-Based Exploration 30

        2.4.1 Overview 30

        2.4.2 Extraction of frontier edges 31

        2.4.3 Limitations 32

    2.5 Mean Shift Clustering 33

        2.5.1 General description 33

        2.5.2 Mean shift using flat kernel functions 35

2.6 Rao-Blackwellized Particle Filters (RBPF) 36
2.6.1 Rao-Blackwellization 36
2.6.2 Mapping with known poses 38
3. RRT-Based Exploration Strategy 39
3.1 Terminologies 40
3.2 Why RRT? 40
3.3 RRT-Based Frontier Detector 41
3.3.1 Local frontier detector 42
3.3.2 Global frontier detector 43
3.3.3 The need for global and local frontier detectors 44
3.4 The Filter Module 45
3.5 Robot Task Allocator 45
3.5.1 Market-based assignment 47
3.5.2 Discount process 48
3.5.3 Hysteresis gain 49
3.5.4 Removing the discount 50
3.5.5 Calculation of information gain 50
3.6 Implementation 51
3.6.1 Mapping and localization module (SLAM) 51
3.6.2 Path planning module 51
3.6.3 Map merging module 53
3.6.4 Global and local frontier detector modules 54
3.6.5 Filter module 55
3.6.6 Robot task allocator module 56
4. Simulation and Experimental Work 58
4.1 Simulation Setup 58
4.1.1 Robot model 58
4.1.1.1 Environments used in simulations 59

4.2 Hardware Setup 60
4.2.1 Laser scanner 61
4.2.2 Robot platform 61
4.2.3 Robot computer 62
4.2.4 Master computer 62
4.2.5 Network setup 63
4.2.6 The map used in the experiment 64
4.3 Results 64
4.3.1 Simulation results 64
4.3.2 Experimental results 67
5. Conclusion and Future Work 69
5.1 Conclusion 69
5.2 Future Work 69
References 71
Appendix A: Raw Data Results 75
Single Robot, Simulation, First (Large) Map Results 75
Single Robot, Simulation, Second (Small) Map Results 76
Single Robot Real Experimental Results 77
Three Robots, Simulation, First (Large) Map Results 78
Three Robots, Simulation, Second (Small) Map Results 79
Three Robots Real Experimental Results 80
Vita 81

List of Figures

Figure 1: Navigation hierarchy 12

Figure 2: Yamauchi frontier detection 15

Figure 3: RRT graph structure 20

Figure 4: The propagation of the tree in the RRT algorithm 22

Figure 5: ROS file system 24

Figure 6: Common coordinate frames associated with mobile robots 27

Figure 7: An occupancy grid map 28

Figure 8: A zoomed view showing the cells of an occupancy grid map 29

Figure 9: A topological map 29

Figure 10: Extraction of topological maps 30

Figure 11: Extraction of frontier edges in an occupancy grid map 31

Figure 12: The process of extracting frontier edges 32

Figure 13: Kernel density estimation using Gaussian kernel function 34

Figure 14: Gaussian kernel function with a larger bandwidth 34

Figure 15: Gaussian kernel function with a smaller bandwidth 35

Figure 16: Kernel density estimation using flat kernel 36

Figure 17: Mean shift using flat kernel functions 37

Figure 18: Overall schematic diagram of the exploration algorithm 39

Figure 19: RRT Voronoi diagram 41

Figure 20: Propagation of the local RRT and the detection of frontier points . . 43

Figure 21: Global and local frontier detectors 44

Figure 22: Information gain region of a frontier point 46

Figure 23: A scenario for robot assignment 48

Figure 24: Updating information gain of a frontier point in the discount step . . 49

Figure 25: Calculation of information gain for a frontier point 51

Figure 26: Implementation diagram 52

Figure 27: Costmap showing how obstacles are inflated 53
Figure 28: Map merging example 53
Figure 29: Map merging limitation 54
Figure 30: Simulated Kobuki platform 59
Figure 31: Simulation environment, first map 59
Figure 32: Simulation environment, second map 60
Figure 33: Overview of the hardware setup 60
Figure 34: Laser scanner mounting on the Kobuki mobile base 61
Figure 35: Robots used in the experiments 62
Figure 36: The real map used in the experiments 64
Figure 37: Occupancy grid maps generated using the proposed exploration strategy, single robot case. 65
Figure 38: Occupancy grid maps generated using the proposed exploration strategy, three robots case. 65
Figure 39: Large map simulation results 66
Figure 40: Small map simulation results 67
Figure 41: Real map results 68
Figure 42: Raw data results for single robot, simulation, first (large) map 75
Figure 43: Raw data results for single robot, simulation, second (small) map . . 76
Figure 44: Raw data results for single robot real experiments 77
Figure 45: Raw data results for three robots, simulation, first (large) map 78
Figure 46: Raw data results for three robots, simulation, second (small) map . . 79
Figure 47: Raw data results for three robots real experiments 80

Chapter 1: Introduction

Robot navigation has gained significant interest in research. For a robot to navigate efficiently (i.e. at minimum cost) through an environment, a map has to be provided. In order to autonomously generate such a map, a robot is required to explore an unknown region. Map exploration can be defined as the act of moving through an unknown environment while building a map to be used for subsequent navigation [1]. This process should be efficient, gaining as much information as possible in the shortest amount of time. Thus, robots should avoid space that is already explored, and be biased toward the unknown space.

Exploration is on the top of the navigation hierarchy, as shown in Figure 1. At the bottom is robot localization followed by path planing in the middle. The ability of the robot to localize itself (i.e. know its current pose, which is defined as the position and the orientation of the robot) is needed for path planning and map construction. In order to plan a path, both target and current poses are needed, and they have to be represented in terms of a fixed global frame. This global frame with frame transformations is needed for building the map because range sensor readings (e.g. from a laser scanner) used in mapping have to be transformed from the sensor frame to the global frame.

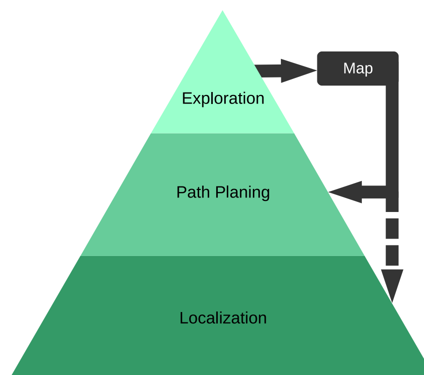


Figure 1: Navigation hierarchy

The second problem in the hierarchy is path planning. The exploration strategy generates target points to be explored in the environment. Robots should be able to use the available generated map, and path plan their way to target points avoiding obstacles.

While doing so, the map has to be updated. The map can also be used in relative localization if no absolute positioning system is used. This approach is referred to as Simultaneous Localization and Mapping (SLAM).

Frontier-based exploration strategies [1–3] have received great attention compared to other exploration strategies. In frontier-based exploration, robots are directed towards frontiers, which are the borders separating explored space from unknown space. Frontier-based exploration strategies use image processing tools to detect frontier edges in a map. Researchers have tried to improve this approach by proposing faster frontier detecting algorithms, and such approaches have been implemented on both single robots and on a team of multiple robots. Other exploration strategies utilize randomized motion planning techniques which relay on their probabilistic nature of being biased toward unexplored space. In both previous exploration classes, the main focus is to reduce the probability that previously explored regions are explored again. This increases efficiency of exploration. Also researchers are working to make the coordination of robots decentralized, so that the exploration algorithm is more robust against individual robot failures. However, if a decentralized robotic exploration algorithm is used, robots then only possess locally available data and information, which comes at the expense of poor coordination and thus can result in reduced efficiency.

This work proposes a map exploration strategy for single and multi-robot exploration. The map exploration strategy is based on a path planning algorithm called the Rapidly-exploring Random Tree (RRT). The RRT algorithm is used to detect frontier points in the map that is being built as robots explore the environment. Using RRT provides a general approach for fast detection of frontier points in a map, because the RRT algorithm is known to be biased towards unexplored regions [4] which biases the tree to detect frontier points. The RRT algorithm also works in 3-D maps. As a result, the proposed algorithm is not limited to detection of frontier points in 2-D maps (as is the case with image processing-based algorithms), and can also be extended to detect frontier points in 3-D maps.

Unlike randomized search techniques, the proposed strategy directs robots to exploration targets without making them follow the edges created by the RRT as it grows. Instead, robots follow a path created independently from the tree. This independence

between the growth of the tree in RRT and robot movements allows the tree to grow faster, increasing the speed of frontier point detection.

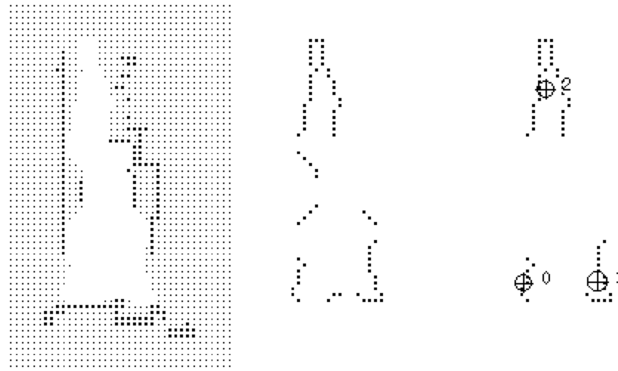
Conserving robot battery capacity is very important when robots explore unknown regions. Similarly, assigning robots target exploration points that are not very far from their current position is also important. This is due to the fact that, if a robot is assigned a point to be explored, which is very far from its current position, then there is a chance that the robot will traverse a lot of pre-explored space. Further, greater motion will result in decreased battery capacity with relatively little information gain. To counter such effects, this work proposes a market-based centralized task allocation procedure. The task allocation procedure relies on RRT to generate commanded points the robot should explore. Further, based on the proximity of a robot to a commanded point, and the expected information gain of the point, the task allocation procedure assigns a robot a target point to be explored.

1.1. Literature Review

Yamauchi [1] was one of the first researchers to use a frontier-based exploration algorithm. It is an algorithm that can autonomously explore and map complicated environments, like offices cluttered with furniture and obstacles.

This algorithm is based on directing the robot towards frontiers, which are "regions on the boundary between open explored space and unexplored space" [1], by which the robot can be directed to unexplored regions increasing the open space, until the whole area is explored. The algorithm was tested in a real office-like environment with a single robot. Figure 2 shows the process of detecting frontier regions, where the robot will attempt to explore the nearest region among them. In the figure, the white region represents known space, the dotted region represents unknown space, and three frontier regions are detected marked by a '+' sign within a circle.

In [2], Yamauchi further applied his algorithm to multiple robots, which work in a cooperative, decentralized manner, and a global map is built from the information shared among the robots. This algorithm may result in duplicate exploration because navigation of each robot operates independently from the others, and the frontier regions



(a) Obtained map (b) Frontier edges (c) Frontier regions

Figure 2: Yamauchi frontier detection. Figure is extracted from [1]

are computed locally on each robot. As a result, two robots might end up exploring the same frontier [3].

Different exploration strategies followed Yamauchi's work, thus there are many exploration algorithms based on the idea of detecting frontiers. In [5] the authors proposed a solution to the problem of duplicate exploration which Yamauchi's algorithm has in the multi-robot case. The solution is based on dividing the region into sub-areas, where each robot will not attempt to enter the sub-area of another robot. A robot will keep exploring its sub-area until no additional frontier region is detected (i.e. the sub-area is completely explored). The robot will then enter into what is known as a "walking state", where the robot tries to find the next sub-area to explore.

In Yamauchi's work [2], frontier detection is conducted each time the robot reaches a target point. This decreases exploration efficiency because the process of detecting frontier edges has a high computational cost [6] due to the fact that the whole map has to be scanned each run. To save computational resources, it is customary to avoid frequent computation of frontiers, in some cases, this can lead to unnecessary redundant exploration tasks [3].

In [3] the authors propose two algorithms for detecting frontiers more quickly. They are the Wavefront Frontier Detector (WFD), and the Fast Frontier Detector (FFD). In WFD, the search for frontiers is done by scanning only the known region of the map. the WFD detector only searches for frontier points, and when a point on a frontier edge is detected, the containing edge can be extracted afterwards. In FFD, only the laser scans

are included in the search. Further, by first converting laser scans into contours, a new frontier is created and stored if the detected frontier does not match any previously stored one.

Another direction of work relates to exploration is based on randomized motion planning techniques. The main focus here is on the RRT algorithm. The main idea behind frontier-based approaches is to ensure that robots explore the unknown space and avoid what has already been explored. This can also be achieved by utilizing the probabilistic features of RRT-based randomized techniques. Robots can be directed to unexplored space without the need for detecting frontier regions. Using the RRT algorithm, the tree of explored nodes formed as a robot moves is biased to expand in the unexplored space [4]. In [7] the authors propose an exploration strategy based on the Sensor-based Random Tree (SRT), which is a variation of RRT. In SRT target exploration goals are generated randomly. These random points are generated within the space surrounding the robot such that all generated points lie within a certain radius of a sensor used to scan the environment. This is a key difference between SRT and RRT, where the latter generates points randomly in the whole map. This makes SRT a depth-first exploration, meaning that a series of random points will be generated more like a chain of nodes, whereas in RRT, tree branches extend in different directions growing subbranches as well. Due to the above-mentioned differences, SRT-based approaches are in need of backtracking, i.e. the robot has to go back and track parent nodes of a current node when a branch stops growing (for example when a robot reaches a dead end). The same place may be visited multiple times during backtracking. Since this is not desirable, researchers have proposed solutions to improve backtracking [8, 9].

In [10], the authors applied the previous SRT-based approach to multiple robots. The SRT-based approach the authors used is a decentralized approach where each robot starts by building its own SRT rooted at a robot's initial pose. After a robot can no longer expand its own tree, it will attempt to help other robots expanding their trees, in what is known as the "supporting phase". The authors further improved this work in [11], by introducing direct shortcut paths between connectable nodes on the tree (called "bridges") which help in the supporting phase, and all robots expand a single tree.

Different exploration algorithms that do not follow the above-mentioned exploration approaches can be classified under information-based exploration [12, 13]. They take into account the localization and mapping problems concurrently with exploration, such that target exploration goals are chosen in a way that reduces the uncertainty of the robot's pose estimate which helps to maximize map information.

An interesting approach for coordinating multiple robots in exploration is proposed in [14]. Multi-robot coordination is based on a market structure, where robots are directed in a way that maximizes revenue and minimizes cost. The gain of information is the revenue, and the expected distance to reach a goal is the cost. Each individual robot tries to maximize its profit, the overall effect is a team of robots with high productivity (i.e. high gain of information and thus more efficient exploration).

In [15] the authors studied the effect of number of robots on exploration efficiency, which they measured in terms of time needed for exploration, and the energy consumed by all robots in the team. The conclusion was that the more robots are added the better. A frontier-based exploration algorithm was used in their research.

1.2. Motivation

Several applications in robotics require the ability of robots to autonomously acquire a map of the environment, where the map is used to navigate efficiently. Search and rescue robots can conduct exploration to map danger zones that cannot be reached by human operators, where a team of robots can first explore the area and provide a map for other robots to use. Another application is cleaning robots. Most cleaning robots in the market randomly traverse a house or a room until eventually the whole area is covered and cleaned; the robot does not know what the house looks like or what its current location is. Using map exploration, cleaning robots can discover the house first and then start the cleaning process systematically.

This work proposes an algorithm for detecting frontier regions in a given map regardless of its dimensionality, as it is applicable to both 2-D and 3-D maps. The proposed algorithm is compared against another commonly used frontier detection

algorithm, and it is shown that the proposed frontier detection algorithm delivers similar performance in 2-D map exploration.

1.3. Thesis Organization

In Chapter 2, a brief background is provided which covers path planning using the RRT algorithm and Dijkstra's algorithm, ROS which is the software platform used in the implementation, map representations, the image-based frontier detection algorithm which is used in the comparison, mean shift clustering, and the mapping and localization algorithm.

Chapter 3 explains the exploration strategy and the implementation. In Chapter 4, the simulation and experimental setups are shown along with the simulation and experimental results. Finally, the report is concluded in Chapter 5.

Chapter 2: Background

This chapter provides some needed background on path planning which includes the RRT algorithm, as well as a brief overview of the Robot Operating System (ROS), as ROS is used to implement and test the map exploration algorithm both in simulation and in the real experimental setup. This chapter also covers different map representations that exist including the occupancy grid map representation which is used in the implementation. Additionally, this chapter covers mean shift clustering, and the Simultaneous Localization and Mapping (SLAM) problem, which are also required in the implementation.

2.1. Path Planning

In a path planning problem, there is an initial point where the robot starts, and a target or a goal point where the robot wants to reach. Both points are located in a map containing obstacles to be avoided. This section covers two path planning algorithms, the RRT algorithm, and Dijkstra's algorithm. RRT is used to find exploration targets, i.e. points that have not been explored yet. Dijkstra's algorithm is used as a path planner for the robots, which finds the shortest path between the robot's current position and the target position.

2.1.1. Common terminologies used in RRT and Dijkstra's algorithm.

Map X: The space containing occupied and free spaces.

Free Space X_{free}: The space that is not occupied by obstacles.

Vertices V: Nodes or points in the map. In RRT these points are connected to one another by the tree branches, each point on the tree is a vertex, vertices are stored in set V.

Edge E: An edge is the branch or line connecting two vertices. Each edge is stored in terms of the spatial coordinates of the two connected points, and edges are stored in the edge set E.

Graph G: Edges and vertices both form a graph, $G = (V, E)$. In RRT the graph has the structure of a tree, as shown in Figure 3.

SampleFree: A function that returns points that are independent identically distributed (i.i.d) in the free space X_{free}.

Nearest $(G = (V,E), x \subset X_{free})$**:** A function that takes a graph (i.e. tree vertices and edges) and a point in the free space as inputs. The function returns the closest vertex to a point $v \subset V$ such that, $\texttt{Nearest}(G = (V,E),x) = argmin_{v \subset V} \|x - v\|$.

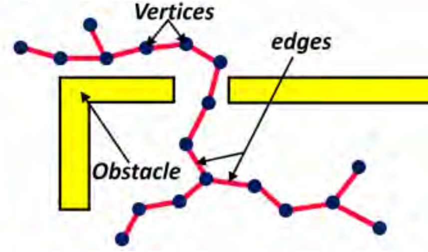


Figure 3: RRT graph structure

Steer: A function that takes two points x and y, and returns a point z, where $\|z - y\|$ is minimized, while $\|z - x\| \leq \eta$, for an $\eta > 0$, where η is the tree growth rate.

ObstacleFree: A boolean function that takes two points in the free space, and returns `True` if there is no obstacle in between.

getFirstMinCost: A function that returns the first single vertex $x \subset G$ carrying the lowest cost.

neighbor(Q,u): A function that returns the set of vertices $X_{neighbor} \subset Q$, where the vertices in this set are directly connected to u.

Cost(x_1, x_2**)**: The cost assigned to an edge connecting two vertices, it can be the length of the edge or an assigned cost.

Parent: Each vertex has a unique parent vertex, and each parent vertex can have multiple child vertices.

2.1.2. The RRT algorithm. RRT was introduced by S. Lavalle [4]. It has a tree structure that starts from a single initial vertex $V = \{X_{init}\}$, and $E = \phi$. At each iteration a random point $x_{rand} \subset X_{free}$ is sampled. The nearest vertex $x_{nearest}$ on the tree to this random point x_{rand} is found, where $x_{nearest} \subset V$.

The **Steer** function then generates a point x_{new} in between $x_{nearest}$ and x_{rand}. If there is no obstacle, both the edge $\{(x_{nearest}, x_{new})\}$ and the vertex x_{new} are added to the

tree. As a result, the tree incrementally grows in the free space until the target point is reached, at which point the algorithm stops. The pseudocode for **RRT** is listed in algorithm 1 [16]. Figure 4 shows the propagation of a path being planned using RRT in a given map.

Algorithm 1 RRT algorithm

```
1: V ← X_init; E ← φ;
2: while x_new ≠ x_target do
3:     x_rand ← SampleFree;
4:     x_nearest ← Nearest(G(V,E), x_rand);
5:     x_new ← Steer(x_nearest, x_rand);
6:     if ObstacleFree(x_nearest, x_new) then
7:         V ← V ∪ {x_new}; E ← E ∪ {(x_nearest, x_new)};
8:     end if
9: end while
10: return G = (V,E);
```

2.1.3. Dijkstra's algorithm. Given a graph **G(V,E)**, Dijkstra's algorithm is used to find the shortest path between any two vertices belonging to **G** assuming such a path. In the hardware implementation of the proposed work, Dijkstra's algorithm is used to direct robots toward a target exploration node, as they explore the space, whereas the target exploration nodes are obtained from a tree generated by the RRT algorithm. For a vertex R on the minimum cost (or distance) path between vertices A and Z, finding this path implies finding the minimum path between A and R. In other words, the optimum path that connects two vertices, consists of optimum paths connecting vertices that belong to it [17]. Thus, the algorithm keeps finding optimum paths to all vertices until the target vertex is visited. The following is a pseudocode of Dijkstra's algorithm [18].

2.2. ROS Background

ROS stands for Robot Operating System, and is a software platform for robots. ROS combines a set of tools and services that are usually provided by any operating system. It was developed with the purpose of maintaining code reusability, so that

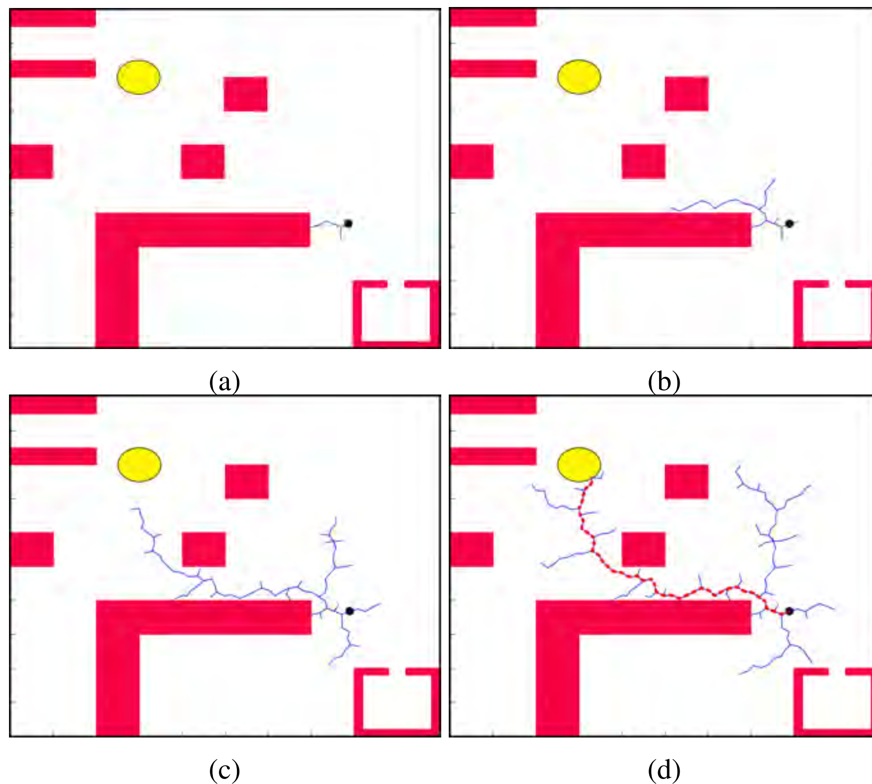


Figure 4: The propagation of the tree in the RRT algorithm

In (a) the tree starts from an initial point, in (b) and (c) the tree keeps growing, in (d) a path (red dashed line) to the target (yellow circle) is found

robotics developers can create hardware-independent software. The tools and services that are provided by ROS include:

- Hardware Abstraction: The software a person writes for a robot is independent of the hardware used. Each sensor or actuator on the robot should have a ROS driver. The driver is responsible for exchanging data with the hardware at the low level, and it wraps the data in a standard message format and passes it to the ROS software that uses the data. The driver is a piece of software that communicates with the hardware. For instance, it can be a C++ program that reads the serial messages sent from a sensor. It can also be a C++ program that sends motor velocity commands as a serial message to a microcontroller, which is connected to the motors and is controlling them.
- Device Drivers: There are many sensors and robot platforms that are supported by ROS. For example, there is a ROS driver for Hokuyo laser scanners, which are commonly used in localization, so the user doesn't have to create a software that

Algorithm 2 Dijkstra algorithm

```
1:  Q ← ϕ;                                          ▷ Create empty vertex set
2:  Parent ← ϕ;                                     ▷ Create empty parent set
3:  for x ⊂ V do
4:      C(x) ← ∞;                                   ▷ Initial cost for all vertices
5:      Q ← Q ∪ {x};
6:  end for
7:  C(x_start) ← 0;                                 ▷ Cost of starting vertex
8:  while Q ≠ ϕ do
9:      C_min ← getFirstMinCost(Q);
10:     u ← x ⊂ Q s.t. C(x) = C_min;    ▷ u gets x, such that cost of x is the minimum
11:     Q ← (Q \ u);                    ▷ Remove current visited vertex from Q set
12:     X_neighbor ← neighbor(Q, u);
13:     for each x_n ⊂ X_neighbor do
14:         cost ← C(u) + Cost(u, x_n);
15:         if cost < C(x_n) then
16:             C(x_n) ← cost;                      ▷ Update vertex's cost
17:             Parent(x_n) ← u;                    ▷ Update vertex's parent
18:         end if
19:     end for
20: end while
21: return Parent, C;
```

extracts data from the laser scanner. Instead, the user only has to install and run the driver. The driver will publish the laser scans in a ROS message format which is standard for all laser scanners.

- ROS Libraries and Community Support: ROS has an active community where researchers around the world share their work. ROS also comes with powerful capabilities for robots, such as path planning, different implementations of SLAM algorithms (Simultaneous Localization and Mapping), and the Adaptive Monte Carlo Localization (AMCL) algorithm.
- ROS file system: ROS software is organized into packages. A package can contain multiple nodes, where each node is a process or a piece of code. A package can also contain message formats, and configuration parameters. In addition to this, each package must have the package manifest, which is a file that describes the package and provides general information about it. ROS packages can also be combined into meta packages (previously they were called stacks). A meta

package usually combines packages that are related to each other. Figure 5 shows the ROS file system.

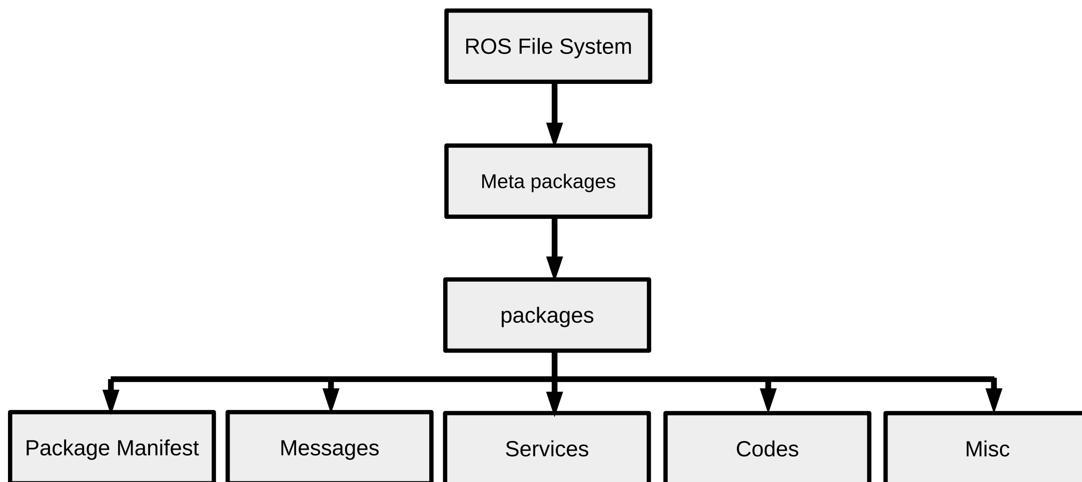


Figure 5: ROS file system

- Message Passing and The Distribution of Computation: ROS provides a message passing interface between processes. Each process can be on a different machine belonging to the same local area network.

2.2.1. ROS concepts. ROS consists of three levels [19]: the file system level (discussed earlier), the computational graph level, and the community level. The computational graph level refers the to the peer-to-peer network of processes which process and share data. The main concepts in this level are listed below:

- Nodes: A node is a process. A ROS node is an executable piece of code that performs certain computations. Typically in a robot there will be multiple nodes running (e.g. a node for controlling the wheels, a node for the laser scanner, and so on). Nodes are located inside packages.
- Publisher: A node that is sending data.
- Subscriber: A node that is receiving data.
- Topics: When a publishing node wants to send data, it puts these data on what is a called a topic, so a node can send data on a certain topic where another node that is subscribing to that topic can receive it. This way subscribers and publishers are

separated, and the communication does not depend on whether there is a node to receive the data. An additional benefit of this is the ability to receive published data by any number of subscribing nodes.

- Messages: Messages are data structures describing the data and their types, nodes send and receive messages over topics. The structure of a ROS message is stored in a simple text file. During the compilation of a package, ROS converts these files into a source code of the programming language used.
  An Example of a ROS message is the ***geometry_msgs/Twist.msg*** message used to send velocity commands to a robot. The ***Twist*** message is stored as a text file with the "*.msg*" file extension, and the file is located under the "*geometry_msgs*" folder that comes with ROS installation. A listing of the ***Twist*** message is shown below:

```
Vector3  linear
Vector3  angular
```

  Where ***vector3*** is a ROS message that has the following structure:

```
float64 x
float64 y
float64 z
```

  "*float64*" is a ROS built-in field type which, for example, in C++ is compiled into a "*double*" data type.
- Master: A master is responsible for node registration and tracking of topics and services. It allows nodes to find each other and it can be thought of as a coordinator.
- Parameter Server: In ROS, parameters (for example configuration parameters, declared constants) are stored by the master in the parameter server. All nodes in a ROS network have access to, and can retrieve stored parameters at runtime.

The last level, the community level, consists of ROS distributions, repositories, ROS forums, and the ROS mailing list.

2.2.2. ROS coordinate frames. Axes orientation in ROS are defined such that the x-axis points to the forward direction of the robot, while the y-axis points to the left, and the z-axis points up [20]. When a project comprises different frames, transformations between them are required. In ROS, frame transformations are computed by the `tf`

package. This package keeps track of the relationships between coordinate frames and stores them in a tree structure buffered in time (when a transformations tree is stored, a time stamp is also captured and stored along with it) [21]. Publishing transformations can be done in two ways, either by writing a node that publishes the transformation, or by a static transform publisher. The former is the general way and works for all transformations, and the latter can be used as a command line tool, or within `roslaunch` files. Some frame conventions related to mobile robots [22] are listed below:

- `map`

  The `map` coordinate frame is a global fixed frame. The pose of a mobile platform represented in this frame is usually estimated using a localization component. Therefore this frame is not continuous and the poses represented in it can jump over time. This frame should not have a significant drift over time, and It is not suitable for local sensing and actuation due to its discrete nature. However, it is useful as a long term reference.

- `odom`

  The `odom` coordinate frame is a global fixed frame. The pose of a mobile platform represented in this frame is obtained from the odometry data. Therefore this frame is continuous but drifts over time, which makes it useful as a short-term reference but not as a global one.

- `base_link`

  This frame is attached rigidly to the mobile robot base, and its origin can be any point on the base. Usually a robot driver publishes the transformation between this frame and the `odom` frame.

- `base_footprint`

  This frame is the projection of the `base_link` frame on the ground, where the pitch and roll angles are zeros, and its yaw rotation is the same as the `base_link` yaw rotation. A static transformation publisher can be used to publish the corresponding transformation.

- `base_laser_link`

  This frame is attached to the laser scanner sensor in which raw laser scans are represented. If the laser scanner is fixed on the robot and does not have relative

motion (which is usually the case), a static transformation publisher can be used to publish the corresponding transformation.

The previous frame names are just conventions and they can have different names. Figure 6 provides a visualization for the above-mentioned frames.

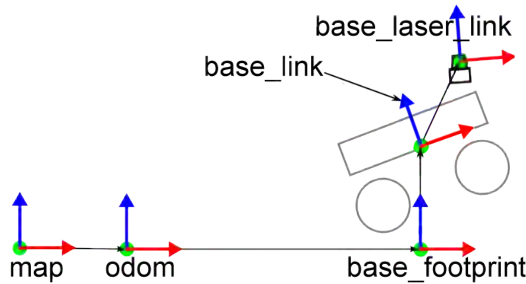


Figure 6: Common coordinate frames associated with mobile robots. Extracted and edited from [23]

2.2.3. ROS launch files.

Launch files are called by the `roslaunch` tool. The roslaunch tool has three main functions: running nodes, names remapping[1], and setting parameters on the parameter server. Instead of running nodes individually, which can be impractical especially in large projects containing many nodes with different configurations, a launch file can run nodes locally and remotely as well. Launch files are also used to set parameters on the parameter server, which is intended for static parameters. Usually, packages include parameter files that are used for node configuration.

2.3. Map Representations

A map is a description of an environment. In general, it consists of a list of objects located in the environment along with their properties [24]:

$$\mathbf{m} = \{m_1, m_2, ..., m_N\} \tag{1}$$

where N is the number of objects in the map. An object can be a landmark, or it can be a certain location in the environment.

[1]Changing a ROS name, this allows running the same node under multiple configurations.

Map representations can be classified under two categories; **feature-based** maps, and **Volumetric** maps.

2.3.1. Volumetric maps. Also known as ***metric maps*** or ***Location-based maps***, in this type of map representation, each object in Equation 1 corresponds to a certain location in the environment. A location-based map includes all locations in the environment; it maps the occupied space where obstacles are located, and the free space where the environment is not occupied. A typical example of location-based maps is the occupancy grid map (see Figure 7).

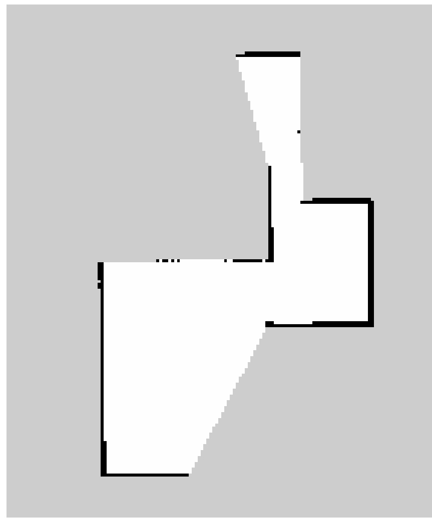


Figure 7: An occupancy grid map

In an occupancy grid map, the environment is represented as a 2-D grid that consists of cells. Each cell carries an occupancy value that refers to the probability that the cell is occupied. An occupancy grid map is expressed as follows:

$$p(\mathbf{m}) = \prod_i p(m_i) \tag{2}$$

where m_i represents a cell in the grid which corresponds to a location in the environment. A probability value $p(m_i) = 0$ means the cell is free, while a probability value $p(m_i) = 1$ means the cell is occupied, and a probability value $p(m_i) = 0.5$ means the state of the cell is totally unknown (maximum confusion). Visualization tools (like "Rviz" software

which comes with ROS) show the occupancy grid as an image where the free space is marked in white, occupied space is marked in black, and unknown space is marked in gray. Map resolution depends on the cell size of the grid (see figure 8).

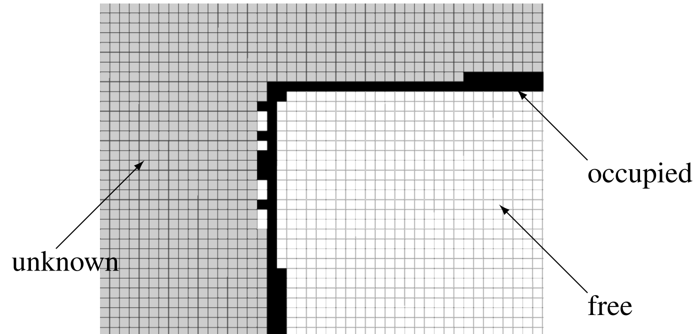


Figure 8: A zoomed view showing the cells of an occupancy grid map

2.3.2. Feature-based maps. Unlike the previous representation, feature-based maps only represent certain features in the environment. Each object m_i in Equation 1 corresponds to a feature, and m_i contains its properties including its location. A feature could be a visual landmark, hence, this type of map representation usually requires vision sensors. Feature-based maps don't store all locations in the environment, and due to that, they are memory-friendly.

2.3.3. Topological maps. Topological maps store only certain places in the environment, along with the relationships between these places. The graph of a topological map contains nodes which represent certain locations in the space, and linking edges between the nodes. See Figure 9.

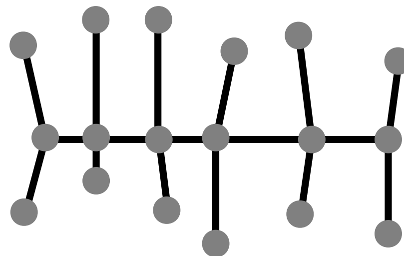
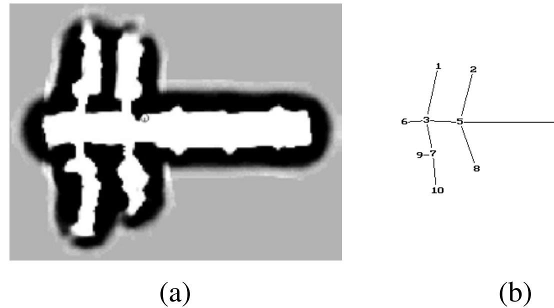


Figure 9: A topological map

Topological maps can be extracted from occupancy grid maps. An example of this is shown in Figure 10.



(a) (b)

Figure 10: Extraction of topological maps

A topological map in (b) extracted from the occupancy grid in (a) [25]

2.4. Frontier-Based Exploration

Frontier-based exploration algorithms can be considered as the most widely used approach for autonomous exploration thus far. Therefore, we use a frontier-based exploration algorithm in the comparison against the proposed exploration algorithm (i.e. RRT-based exploration).

In the RRT-based algorithm, RRT is used to find exploration targets, and the robot task allocator is responsible for assigning the detected exploration targets to each robot. So in order to be able to compare how each algorithm detects target points, our own frontier-based algorithm is implemented such that it only detects exploration goals, and does not allocate them to the robots. By this configuration, both RRT and the frontier-based algorithms use the same task allocator for robot assignment. This eliminates the effect of task allocation so only the detection of exploration targets is compared across both approaches.

2.4.1. Overview. The frontier-based algorithm detects exploration goals by extracting frontier edges. In an occupancy grid map, frontier edges are the lines separating known space (marked in white) and unknown space (marked in gray). After extracting the edges, the center of each edge is marked as an exploration target.

2.4.2. Extraction of frontier edges.

In our implementation, frontier edges are extracted using image processing tools provided by the Open Source Computer Vision (OpenCV) library. The process of extracting frontiers is explained below:

1. A ROS occupancy grid map message is converted into a gray-scale image format. This is needed in order to be able to use OpenCV tools. The occupancy grid message has the following structure [26]:

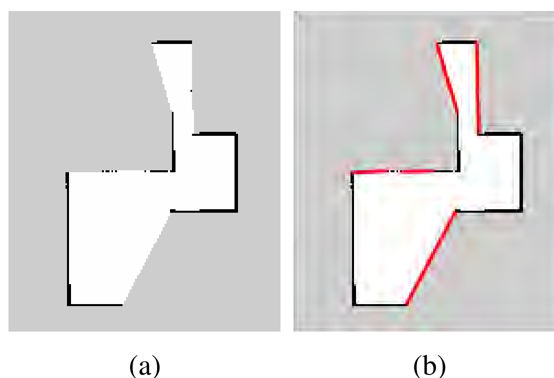
```
Header header
MapMetaData info
# The map data, in row-major order, starting with (0,0).
   Occupancy
# probabilities are in the range [0,100].  Unknown is -1.
int8[] data
```

   where "data" is a 1-D array, and its elements are the occupancy values of each grid in the map. A gray-scale image in OpenCV is stored as a 2-D array, and each element in the array caries the color of the corresponding pixel. The "data" array is converted to an image by reordering its elements and storing them in a 2-D array. Each element is converted to an appropriate color value as follows:

   grid cell value of 0 (unoccupied)$\rightarrow$ pixel with a color value of 255 (white)

   grid cell value of 100 (occupied)$\rightarrow$ pixel with a color value of 0 (black)

   grid cell value of -1 (unknown)$\rightarrow$ pixel with a color value of 205 (gray)



(a) (b)

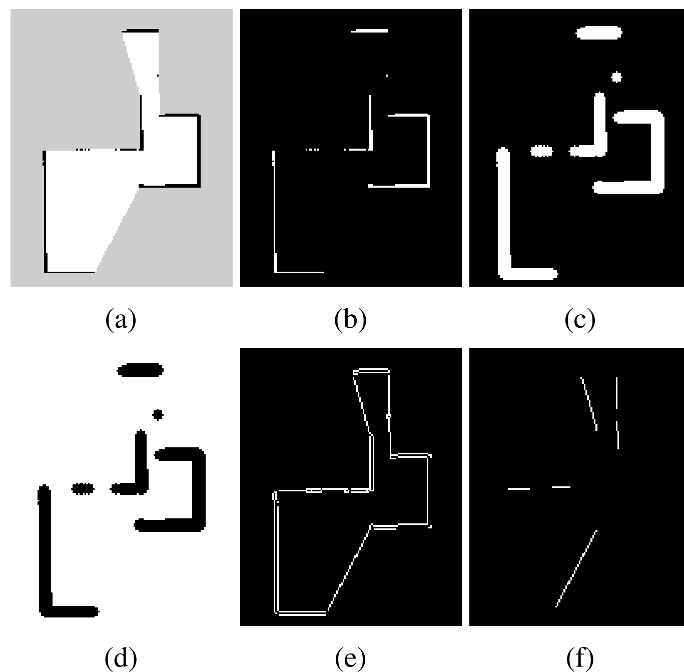
Figure 11: Extraction of frontier edges in an occupancy grid map

An occupancy grid map is shown in (a). In (b) frontier edges are extracted and marked in red

2. A threshold is applied on the image to only keep the pixels that correspond to occupied cells (figure 12b), contours are then drawn and marked on the image (Figure 12c). After that, image colors are inverted (Figure 12d). The result is an image that contains only the occupied grids marked in bold black lines.

3. The next step is applying the Canny edge detector, which returns an image containing all the edges including the occupied grids (the obstacles or the walls), and the frontier edges (Figure 12e).

4. The last step is to subtract the occupied cells from the edges obtained in the previous step. This will keep only the frontier edges. It is accomplished by a bitwise AND operation between the images obtained in the previous two steps. The final result is shown in Figure 12f.



(a) (b) (c)

(d) (e) (f)

Figure 12: The process of extracting frontier edges

2.4.3. Limitations. Although frontier-based algorithms have shown success in autonomous exploration, they have a significant limitation in that they are highly dependent on the map representation. In particular, they need the map to be represented

as an occupancy grid. As a result, frontier-based algorithms are more suitable for 2-D exploration. For 3-D maps, frontier extraction becomes more complex, and image processing tools, which are used to extract frontiers, cannot be used [27, 28].

2.5. Mean Shift Clustering

This section covers the mean shift clustering algorithm, which is used in the implementation of the exploration strategy in order to reduce the computational load. The exploration strategy, explained thoroughly in Chapter 3, detects frontier points in a given map. The number of detected frontier points can be very large. Usually, a large portion of the detected frontier points are located very close to each other. Hence, frontier points must be clustered to discard redundant frontier points; otherwise, accounting for many detected frontier points would unnecessarily slow down the exploration algorithm without much information gained, since points in the same vicinity correspond to the same frontier edge.

The mean shift clustering algorithm is chosen because it does not require the number of clusters as an input, it only requires specifying the size of the cluster. These features makes it a good fit for our proposed exploration algorithm.

2.5.1. General description.

For a given set of points $\{x_i\}_{i=1}^{N} \in \mathbb{R}^d$, the mean shift algorithm considers the points as samples taken from a probability density function (PDF) $f_K(x)$, where higher probability values correspond to dense regions. The local maxima (or the modes) of $f_K(x)$ correspond to the centers of clusters (it's a multi-modal PDF). The mean shift algorithm iteratively shifts each point towards the modes of $f_K(x)$.

The PDF $f_K(x)$ is obtained by kernel density estimation using a certain kernel function $K(x)$. For example, Figure 13 shows a 1-D data set ($d = 1$), where $f_K(x)$ is obtained using a Gaussian kernel function.

$$f_K(x) = \frac{1}{Nh^d} \sum_{i=1}^{N} K\left(\frac{x - x_i}{h}\right) \tag{3}$$

$$K(x) = \frac{1}{(2\pi)^{d/2}} \, e^{-1/2 \, |x|^2} \tag{4}$$

Equation 3 shows that $f_K(x)$ is the addition of individual kernel functions. In Figure 13, kernel functions are Gaussian distributions centered at the points with a variance of $h = 1$.

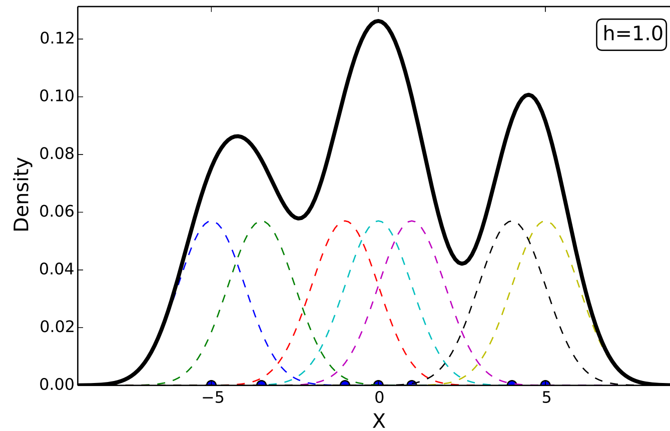


Figure 13: Kernel density estimation using Gaussian kernel function

The kernel function parameter h, is referred to as the bandwidth. For a Gaussian kernel function, h is the variance (the width of the Gaussian distribution). In Figure 13, $h = 1$, and the number of local maxima (modes) is 3 (which is equal to the number of clusters).

Figure 14 shows a kernel function of a larger bandwidth $h = 10$. The number of obtained clusters is 1.

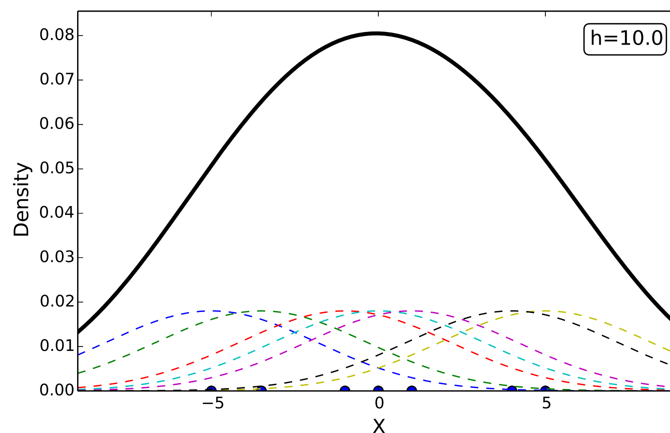


Figure 14: Gaussian kernel function with a larger bandwidth

Figure 14 shows a kernel function of a smaller bandwidth $h = 0.1$. The number of obtained clusters is 7 (which is equal to the number of data points).

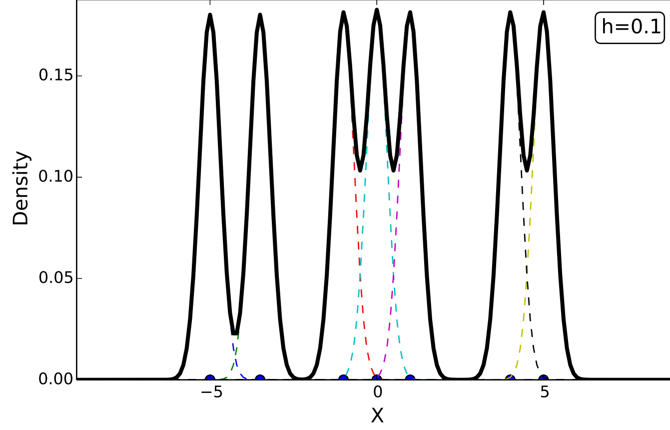


Figure 15: Gaussian kernel function with a smaller bandwidth

The mean shift algorithm shifts each data point in the direction of increasing probability. Hence, for a Gaussian kernel function, it shifts data points in the direction of the gradient $\bigtriangledown f_K(x)$. Shifting a point means adding the mean shift vector (vector of dimension d) to the point (for a Gaussian kernel function, the mean shift vector is calculated from the gradient).

2.5.2. Mean shift using flat kernel functions. In the implementation of the exploration strategy, a Python ready-made library is used [29]. This library includes an implementation of the mean shift clustering that uses a flat kernel function of the following form:

$$K(x) = \frac{1}{2}\begin{cases} 1, & \text{if } ||x|| \leq h \\ 0, & \text{if } ||x|| > h \end{cases} \tag{5}$$

Figure 16 shows the obtained PDF $f_K(x)$ using the flat kernel function on the same data set used in Figure 13, with a bandwidth of $h = 1$.

Each point $x \in \{x_i\}_{i=1}^N$ is shifted towards the mean $m(x)$ of data points $\{x_i\}_{i=1}^N$ that lie within radius h from the point x [30]. The points are iteratively shifted until they settle and converge to a value. This value is the cluster center, and all the points that

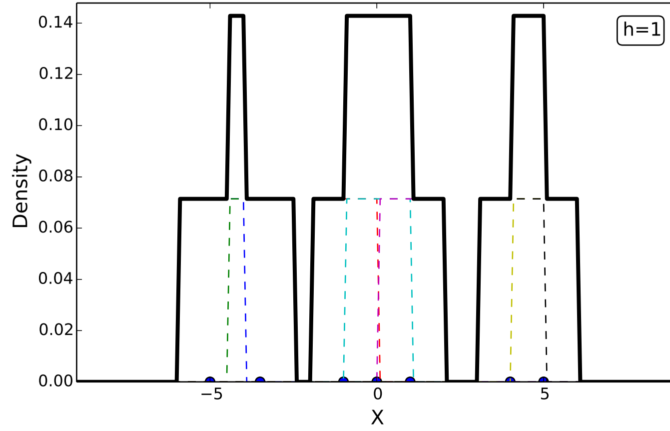


Figure 16: Kernel density estimation using flat kernel

converge to the same center belong to the same cluster.

$$The\ mean = m(x) = \frac{\sum_{i=1}^{N} K(x_i - x)x}{\sum_{i=1}^{N} K(x_i - x)} \tag{6}$$

where "$m(x) - x$" is the mean shift. Each iteration the mean $m(x)$ is computed for each data point $x \in \{x_i\}_{i=1}^{N}$, and each point x is shifted to its associated mean (i.e. $x \leftarrow m(x)$) (where $m(x)$ is calculated using Equation 6). Figure 17 shows how points are shifted each iteration until they settle in the centers of clusters .

2.6. Rao-Blackwellized Particle Filters (RBPF)

The exploration strategy includes a module that is responsible for localizing the robot and building a map of the environment. This problem is referred to as Simultaneous Localization and Mapping (SLAM). For this purpose, a ready-made ROS package is used in the implementation [31], this package implements a SLAM algorithm that uses a Rao-Blackwellized Particle Filter (RBPF). The algorithm takes range measurements from the laser scanner and the odometry of the robot as inputs. The outputs are the robot's pose and a map of the environment represented as an occupancy grid (discussed previously). Occupancy grid maps are discussed below.

2.6.1. Rao-Blackwellization. For any SLAM algorithm, the main goal is to estimate the map and robot's pose within this map. The SLAM problem is formulated as

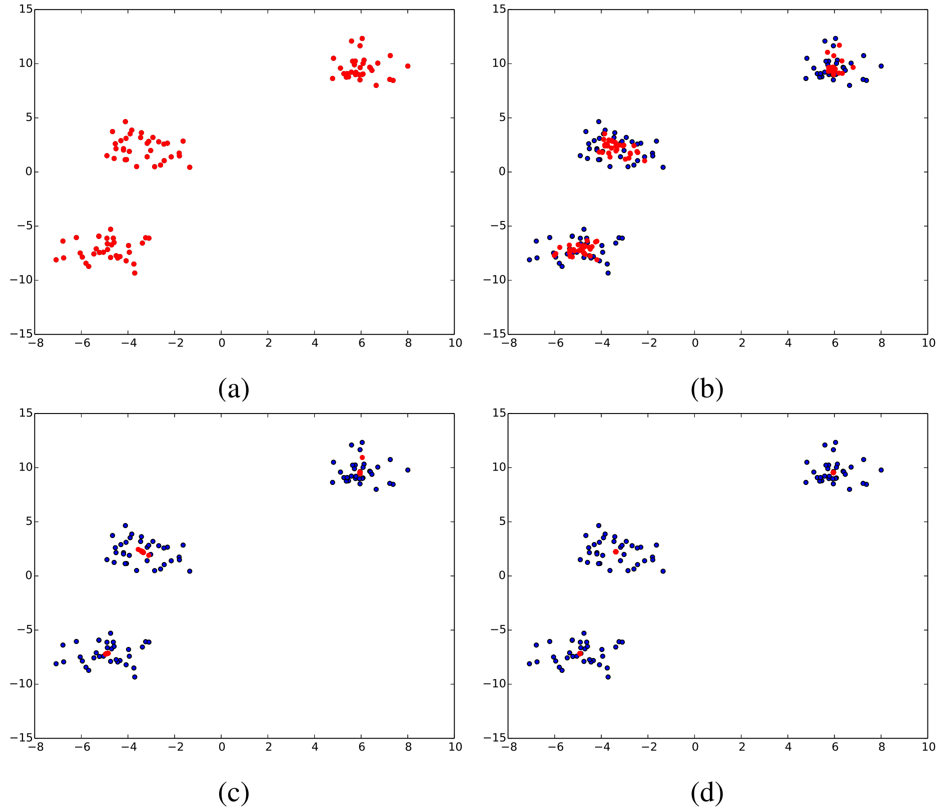


Figure 17: Mean shift using flat kernel functions

In (a) each point initially is a center of a cluster. In Figures (b),(c) for each point the mean of the points lying within a circle of radius h is computed, and each point is shifted to the mean. In (d) the means of all points settle at cluster centers.

finding the following joint probability:

$$p(x_{1:t}, m \mid z_{1:t}, u_{1:t}) \tag{7}$$

where u is the control input (usually taken from the odometry), z is the observation (e.g. from a laser scanner), m is the map, and x is the robot's pose. The RBPF SLAM splits the problem into two independent subproblems: i) estimating the robot's pose posterior $p(x_{1:t} \mid z_{1:t}, u_{1:t})$ using a particle filter, and ii) estimating the map posterior $p(m \mid x_{1:t}, z_{1:t})$ for each particle. The latter subproblem is referred to as "mapping with known poses". Thus the SLAM problem is simplified as follows:

$$p(x_{1:t}, m \mid z_{1:t}, u_{1:t}) = p(m \mid x_{1:t}, z_{1:t}).p(x_{1:t} \mid z_{1:t}, u_{1:t}) \tag{8}$$

This technique is referred to as Rao-Blackwellization [32].

2.6.2. Mapping with known poses. Given a robot's path and the observations, the problem is formulated as follows:

$$p(m \mid x_{1:t}, z_{1:t}) = \prod_i p(m_i \mid x_{1:t}, z_{1:t}) \tag{9}$$

where m is the map (occupancy grid map), and m_i is a cell in this map. Using Bayes filter, the probability $p(m_i)$ can be estimated as follows [24]:

$$l(m_i \mid x_{1:t}, z_{1:t}) = l(m_i \mid z_t, x_t) + l(m_i \mid z_{1:t-1}, x_{1:t}) - l(m_i) \tag{10}$$

where:

$$l(x) = log \frac{p(x)}{1 - p(x)} \tag{11}$$

This equation has three terms. $l(m_i)$ is the prior and is computed from $p(m_i)$ as shown in Equation 11. Normally it is initialized as $p(m_i) = 0.5$ which indicates that the cell is unknown before incorporating any observations. The term $l(m_i \mid z_{1:t-1}, x_{1:t})$ is a recursive term, and $l(m_i \mid z_t, x_t)$ is referred to as the "inverse sensor model", where $l(m_i \mid z_t, x_t)$ is updated from the perception sensor, which in our case is the laser scanner [24].

Chapter 3: RRT-Based Exploration Strategy

The exploration strategy is split into three modules: the RRT-based frontier detector module, the filter module, and the robot task allocator module. The frontier detector module is responsible for detecting frontier points and passing them to the filter module. The filter module clusters the frontier points and stores them. In this work the mean shift clustering algorithm [33] is used. The filter module also deletes invalid and old frontier points (invalid and old frontier points are defined in Section 3.1). The task allocator module receives the clustered frontier points from the filter module, and assigns them to a robot for exploration. Each module is explained in more detail in Sections 3.3, 3.4, and 3.5. The exploration strategy also requires mapping and path planning modules, which are available in existing work [17,32,34]. An overall high level schematic diagram of the exploration strategy is shown in Figure 18. Implementation details are available in Section 3.6.

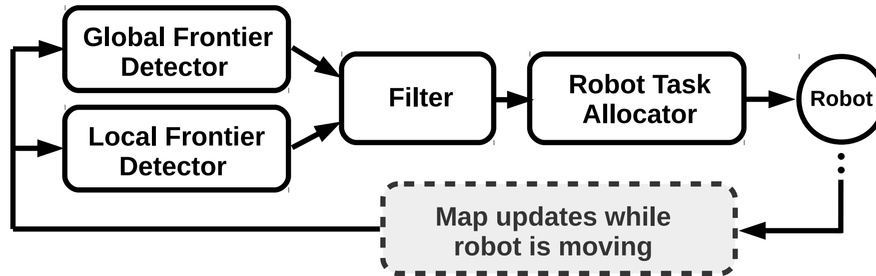


Figure 18: Overall schematic diagram of the exploration algorithm

The proposed configuration, i.e. splitting the exploration strategy into three modules (i.e. frontier detector module, filter module, and task allocator module) has the following advantages. The task allocation routine can be changed without affecting the detection of frontier points. Similarly different types of frontier detectors could be tested using the same robot task allocator (this is required when a comparison between different types of frontier detectors is made). Additionally, multiple instances of the frontier detector can be run in parallel for faster detection of frontier points.

Normally, each robot runs an instance of the local frontier detector module. The master machine runs an instance of the global frontier detector module in addition to the

filter and the robot task allocator modules. The master machine can be one of the robots. So for n robots, there are $n+1$ instances of frontier detector modules, a single instance of the filter module, and a single instance of the robot task allocator module.

This configuration also provides scalability, as computational load can be distributed among the robots, such that each robot can run its own local frontier detector.

3.1. Terminologies

The following terminologies, in addition to the terminologies defined in section 2.1, are needed:

GridCheck: A function that takes a map and two points as arguments. It returns 0 if there is an obstacle between the points based on the given map. It returns -1 if there is an unknown region between the points. Otherwise it returns 1 indicating that the points are in known free space.

PublishPoint: A function that sends detected frontier points to the filter. The 'filter' is described later in the chapter.

Old frontier point: A frontier point detected in earlier iterations of the frontier detector, and which is no longer in the unknown region of the map.

Invalid frontier point: A frontier point the robot cannot physically reach, i.e. no valid path exists between the robot's position and the given frontier point.

3.2. Why RRT?

RRT is heavily biased to grow towards unknown regions of the map [4], which biases the tree to detect frontier points. This property can be explained by showing the Voronoi diagram of RRT during exploration. This diagram shows how the tree is biased to grow and extend from the vertices that are at the ends of the tree.

For a given graph that consists of vertices and edges (similar to the RRT graph), a Voronoi diagram divides the space into regions, where each region is associated with a vertex. A region is a collection of points that are nearest to the corresponding vertex. Thus, a Voronoi diagram can be used to graphically know which vertex is nearest to any given point in the space.

Figure 19 shows the Voronoi diagram of RRT during exploration. Vertices with larger Voronoi regions are closer to the unknown space. Since the selection of vertices is based on finding the nearest neighbor, the tree is more likely to extend from these regions.

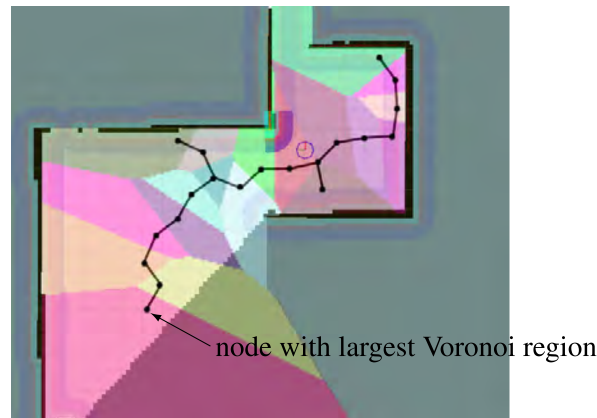


Figure 19: RRT Voronoi diagram

RRT is also not limited to 2-D space. As a result it can be used in 3-D exploration (which can help to extend the work in [35, 36]), unlike image processing techniques which rely on having a 2-D map of the environment. RRT is also probabilistically complete [16], and it is guaranteed that the map will be completely discovered and explored.

3.3. RRT-Based Frontier Detector

The RRT-based frontier detector module discovers frontier points. In our work, any point that is reached by the growing RRT tree is considered a frontier point, if this point lies in the unknown region of the map. In our implementation, the map is represented as an occupancy grid (discussed in Section 2.3). Points located in the unknown region carry a cell value of -1, so by reading the cell value of a point, it can be classified according to its region.

We propose the use of two versions of frontier detectors: i) a local frontier detector, and ii) a global frontier detector. The local frontier detector is meant to be run by each robot. The global frontier detector is meant to be run by the master, which can be

one of the robots. Running additional instances of the local or the global frontier detectors enhances the detection of frontier points at the expense of increasing computational load.

3.3.1. Local frontier detector. The outline is listed in Algorithm 3. Similar to the RRT algorithm discussed in Section 2.1, the tree in the local frontier detector starts from a single initial vertex $V = \{x_{init}\}$, and $E = \phi$, and at each iteration a random point $x_{rand} \subset X_{free}$ is sampled. The first vertex of the tree which is nearest to x_{rand} is found (this point is called $x_{nearest} \subset V$). Then, the **Steer** function generates a point x_{new}. The **GridCheck** function checks if x_{new} lies in the unknown region, or if any point on the line segment between x_{new} and $x_{nearest}$ lies in the unknown region. If either of the above conditions is true, then x_{new} is considered as a frontier point. The point x_{new} is then sent to the filter module, and the tree is reset, i.e. tree nodes and edges are deleted. The next iteration of the tree starts from the current robot position (i.e. $V = \{x_{current}\}$, and $E = \phi$). This step is shown in line 8 of Algorithm 3. If there is no obstacle at x_{new} and no obstacle in the space between x_{new} and $x_{nearest}$, the tree extends by adding x_{new} as a new vertex. An edge is created between x_{new} and $x_{nearest}$.

Algorithm 3 Local Frontier Detector

```
1:  V ← x_init; E ← φ;
2:  while True do
3:      x_rand ← SampleFree;
4:      x_nearest ← Nearest(G(V,E), x_rand);
5:      x_new ← Steer(x_nearest, x_rand);
6:      if GridCheck(x_nearest, x_new) = −1 then          ▷ Unknown region
7:          PublishPoint(x_new);
8:          V ← x_current; E ← φ;                         ▷ Reset tree
9:      else if GridCheck(x_nearest, x_new) = 1 then      ▷ Free Known region
10:         V ← V ∪ {x_new}; E ← E ∪ {(x_nearest, x_new)};
11:     end if
12: end while
```

The resetting of the tree as shown in line 8 of Algorithm 3 is one of the major differences between the usage of RRT for exploration in this work, compared to other standard implementations of RRT available in the literature. The reason behind resetting the tree in the local frontier detector is explained in subsection 3.3.3.

For each robot running the local frontier detector, a tree is generated through the process described above. Once the tree reaches an unknown region, a frontier point is marked and the tree is reset. This process happens during a robot's motion. Therefore the tree grows from a new initial point each time it resets according to line 8 of Algorithm 3. The local frontier detector is proposed for fast detection of frontier points in the immediate vicinity of the robot at any time. Figure 20 shows the propagation of the local RRT and the process of detecting a frontier point.

3.3.2. Global frontier detector. The outline is listed in Algorithm 4. This version of the detector is identical to the local frontier detector with one difference being that the tree doesn't reset and keeps growing during the whole exploration period (i.e. until the map is completely explored), which makes the global frontier detector algorithm similar to RRT. The global frontier detector is meant to detect frontier points through the whole map and in regions far from the robot.

The global frontier detector uses the global map obtained by merging the local maps of all the robots, whereas the local frontier detector uses the local map of a robot.

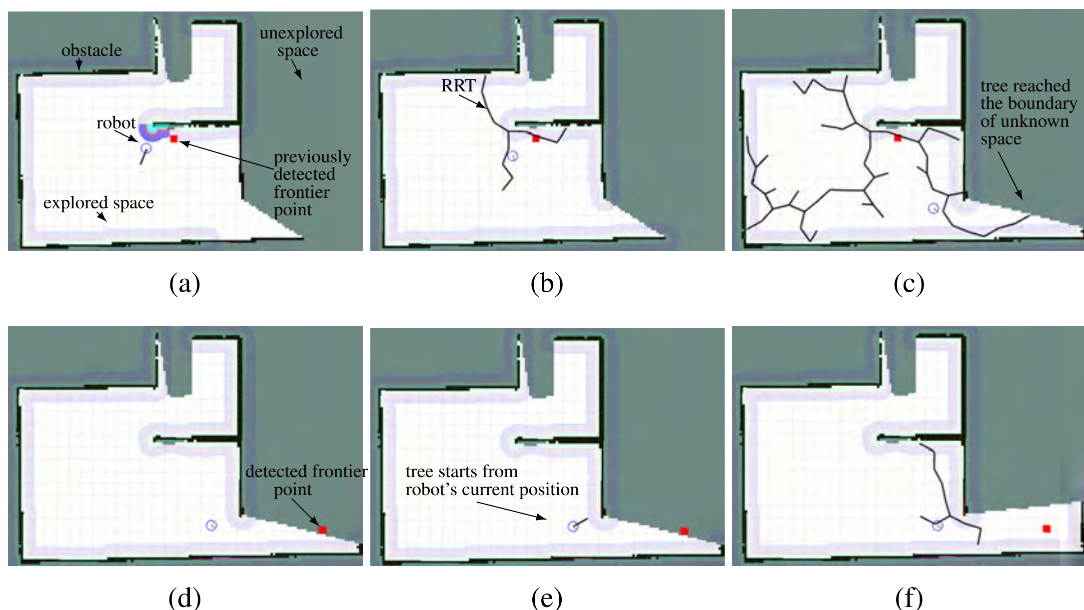


Figure 20: Propagation of the local RRT and the detection of frontier points

In (a) the tree starts from the current position of the robot. In (b) and (c) the tree keeps growing, and in (d) a tree vertex lying in the unknown region is marked as a frontier point and the tree is reset. In (e) and (f) the loop repeats and the tree grows back again from the robot's current position

Algorithm 4 Global Frontier Detector

```
1:  V ← x_init; E ← φ;
2:  while True do
3:      x_rand ← SampleFree;
4:      x_nearest ← Nearest(G(V,E), x_rand);
5:      x_new ← Steer(x_nearest, x_rand);
6:      if GridCheck(x_nearest, x_new) = −1 then          ▷ Unknown region
7:          PublishPoint(x_new);                           ▷ Tree does not reset
8:      else if GridCheck(x_nearest, x_new) = 1 then      ▷ Free known region
9:          V ← V ∪ {x_new}; E ← E ∪ {(x_nearest, x_new)};
10:     end if
11: end while
```

If there is a delay in obtaining the merged global map, the local frontier detectors are still able to detect frontier points.

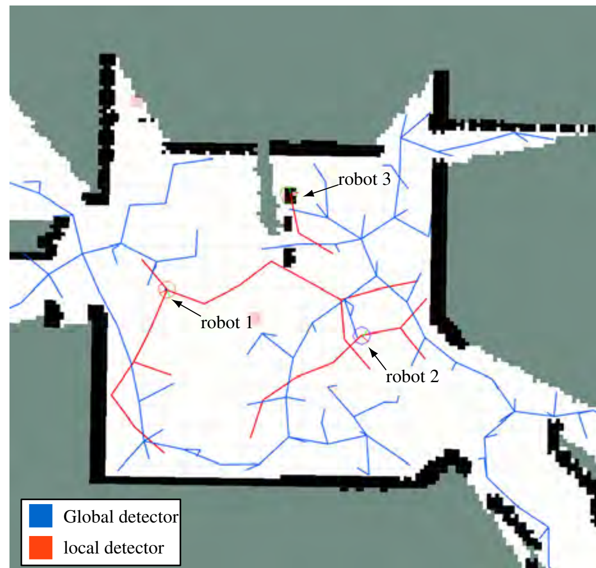


Figure 21: Global and local frontier detectors

3.3.3. The need for global and local frontier detectors. As explained above, the local tree is reset after it detects a frontier point and starts growing again from the robot's current position. This has two consequences. First, it allows quicker detection of frontier points because the tree starts growing from the previously-detected frontier point, so the chance that the next point picked from the RRT for exploration lies in the unknown space is higher. Second, the robot can miss exploring small corners in a map.

To fix this problem, and also to make sure that points which are far from the robot's current position are detected and explored, we use the global frontier detector.

However, the growth of the tree in the global frontier detector becomes slower as the tree grows larger (i.e. the number of tree vertices increases). This can be explained by analyzing the Voronoi diagram of RRT: as the number of tree vertices increases, the space is decomposed into smaller and smaller Voronoi regions. As a result, the **steer** function will create edges of smaller length; hence, detection of frontier points also becomes slower. This is why the local frontier detector is needed. Thus our proposed strategy uses local and global RRT-based frontier detectors to complement one another so that frontier detection is as fast as possible.

3.4. The Filter Module

The filter module receives the detected frontier points from all the local frontier detectors and from the global frontier detector. Every time a new frontier point is received, the filter module stores it in the frontier points array. At first, the filter module clusters the points stored in the frontier points array using the mean shift clustering algorithm [29, 33] discussed in Section 2.5, and it updates the array by discarding (deleting) every frontier point that is not a center of a cluster. The frontier points array is then sent to the robot task allocator module. The clustering and subsequent discarding process is needed to reduce the number of frontier points, since global and local frontier detectors can provide too many frontier points which are extremely close to each other. If such points are sent to the robot task allocator module, then there will be unnecessary consumption of computational resources, and no additional information about the map is necessarily gained. The filter module also deletes invalid and old frontier points in each iteration.

3.5. Robot Task Allocator

This module receives the frontier points array from the filter module and assigns them to the robots. The robot task allocator module assigns frontier points to be explored by a particular robot by considering the following:

- **Navigation cost (N):** The expected distance to be traveled by the robot to reach a frontier point. In order to simplify computation, the navigation cost is calculated by considering the norm of the difference between the robot's current position and the location of a frontier point.
- **Information gain (I):** The area of an unknown region expected to be explored for a given frontier point. The information gain is quantified by counting the number of unknown cells surrounding a frontier point within a user defined radius. This radius is referred to as the information gain radius, which should be set to a value equal to the perception sensor range. The area is then calculated by multiplying the number of cells within the information gain radius, by the area of each cell (which is computed from the map resolution). In Figure 22, the information gain approximately equals 1.81 m^2 (i.e. the number of unknown cells is 181, each cell is a square with a width equal to map resolution, which, in this case, is 0.1 m).

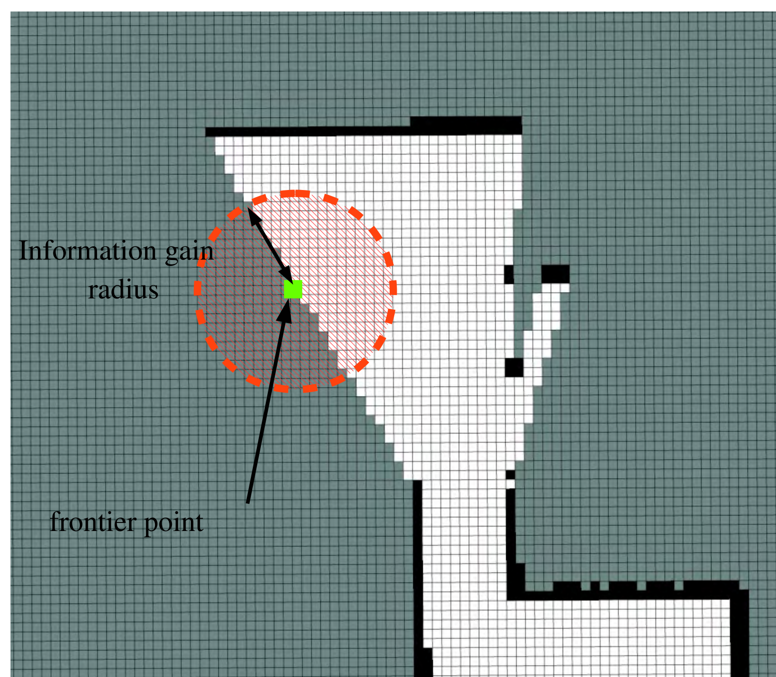


Figure 22: Information gain region of a frontier point

- **Overlapping:** The following must be achieved in order to reduce overlapping: i) a robot should not explore a frontier point that is currently assigned to another robot, or a frontier point that is close to the current location of another robot, ii)

robots should be biased to explore neighbor frontier points which lie in the region surrounding the last assigned frontier point or surrounding the robot's current position.

In order to achieve these criteria, a market based approach is adopted for robot assignment similar to [37] with a few modifications. The outline of the task allocator is listed in Algorithm 5. The following sections explain each step in more detail.

3.5.1. Market-based assignment. The robot task allocator module is run on the master At each iteration a bidding list is computed, where a bid is equal to the expected revenue. The revenue is the expected information gain of exploring a frontier point minus the expected cost to reach that point, and it is computed as shown in lines 14-18 of Algorithm 5. The robot with the highest bid is assigned to the associated point. A bidding list consists of records, where each record is a robot-point combination along with the expected revenue for the robot to explore the point. A scenario for robot assignment is shown in Figure 23. The bidding list created for this scenario is shown below, where it has six records (each robot has two records, because there are two frontier points detected):

```
{'robot_id': 2, 'point': [-4.24, -0.68], 'revenue': 26.24},
{'robot_id': 1, 'point': [-4.24, -0.68], 'revenue': 26.22},
{'robot_id': 2, 'point': [-1.67, -1.94], 'revenue': 6.97},
{'robot_id': 3, 'point': [-4.24, -0.68], 'revenue': 6.60},
{'robot_id': 1, 'point': [-1.67, -1.94], 'revenue': 6.42},
{'robot_id': 3, 'point': [-1.67, -1.94], 'revenue': -1.77}
```

The winning bid is for Robot 2 with the frontier point $[-4.24, -0.68]$. Although point $[-1.67, -1.94]$ is closer to Robot 2, its information gain is small, and hence its revenue is less profitable.

In line 18, the revenue is calculated as shown in Equation 12:

$$Revenue = (\lambda \times I) - N \tag{12}$$

where λ is a positive user-defined constant which is used as a weight. The weight λ is used to give more importance to the information gained from exploring a frontier point,

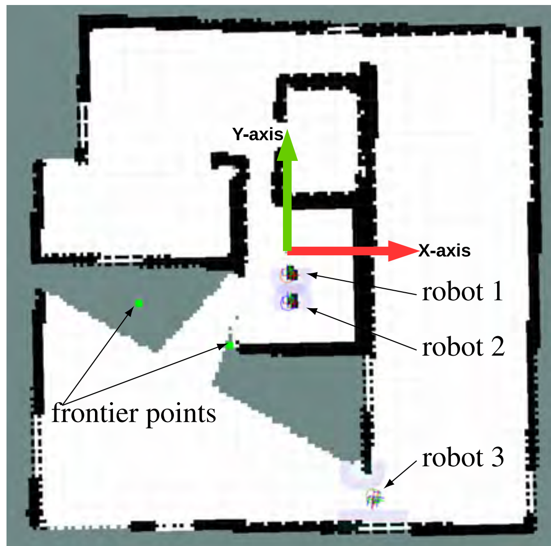


Figure 23: A scenario for robot assignment

compared to the navigation cost. This also helps to ensure that the terms I, and N have a similar order of magnitude.

The calculation of bids and the subsequent robot assignment involves only available robots (robots that are not currently executing an exploration command, and are ready to be assigned). If there are no available robots (all robots are busy), a bidding list is created for all the busy robots, so that if a frontier point with a higher revenue exists, it will be assigned instead of the currently assigned frontier point. An example is when the assigned frontier point is explored before the robot reaches the point. In this scenario, the information gain of the point will be zero, and hence it would be more efficient if a new frontier point with higher revenue is assigned to the robot instead.

3.5.2. Discount process. To avoid assigning the same frontier point to multiple robots, the "`discount`" function is used (line 7 of Algorithm 5). For each frontier point that has an overlapping area with a previously assigned frontier point, the "`discount`" function is applied, where it subtracts that overlapping area from its information gain. Overlapping area is computed by counting the number of overlapping cells and multiplying the count by the area of each cell (which depends on the map resolution). Overlapping cells are the ones common between the information gain region of the previously assigned frontier points, and the current frontier point. Figure 24 shows an example of the discount process.

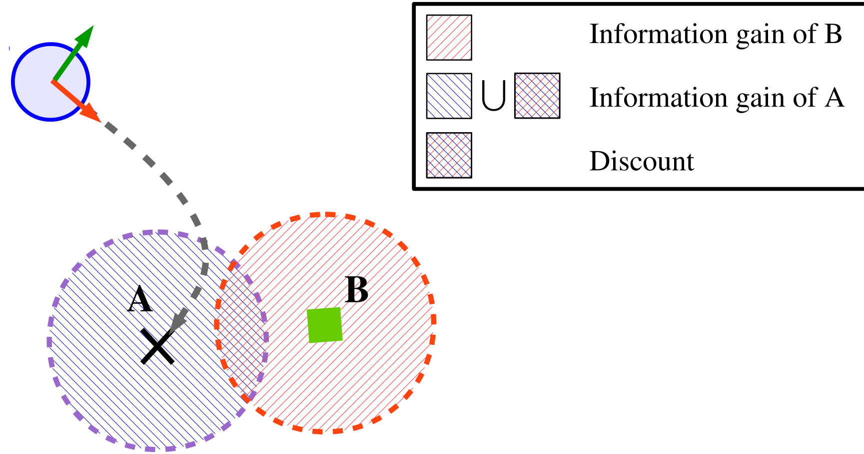


Figure 24: Updating information gain of a frontier point in the discount step

A robot is assigned to point A, and as a result, a discount equal to the overlapping area is applied on the information gain of frontier point 'B'.

3.5.3. Hysteresis gain. The information gain of every frontier point that is close to a robot's current position or a robot's currently assigned frontier point (in case if a robot is currently executing an exploration command) is multiplied by the hysteresis gain (h_{gain}) [37], where h_{gain} is a gain larger than 1 (in our implementation it is set to 2). The radius that defines how close a point should be is referred to as the hysteresis radius (h_{rad}), and is set based on user experience (in our implementation it is set to 3 m).

Applying the hysteresis gain increases the revenue of frontier points that lie in the vicinity of a robot which biases the robot to continue exploring the region surrounding it. This reduces overlapping, as robots do not switch to a new region unless the current region is completely explored. The process of a robot switching between distant regions in the environment is a major source of overlapping, since robots during this process cover pre-explored areas. This is not preferable as it consumes time and cost (where cost is the total distance traveled by a robot).

The hysteresis gain is applied in two cases: i) for available robots, it is applied to the information gain of frontier points that are close to the robot's position within h_{rad} (lines 15-17 of Algorithm 5), and ii) for busy robots (only when there are no available robots). In this case, the hysteresis gain is applied to the information gain of frontier points that are close to the robot's position or close to the robot's currently assigned frontier point within h_{rad} (lines 30-35 of Algorithm 5).

3.5.4. Removing the discount.

Note that the robot task allocator first assigns available robots, and only if all robots are busy, the task allocator reassigns busy robots to frontier points that have higher expected revenue, if such points exist. The discount process reduces the information gain of the frontier points that are near to an assigned point. After applying the discount process, the current assigned frontier point will have zero information gain. Hence, its expected revenue will become small so that other robots will not be assigned this point, which solves the problem of assigning the same frontier point to multiple robots.

However, when all robots are busy, if the discount is not removed, once a robot is assigned a frontier point, its information gain will become zero after the discount. As a result, the robot task allocator will assign a different frontier point to the robot even if this point has a lower revenue. This will cause busy robots to be reassigned different frontier points even before exploring them.

To solve this problem, and when calculating the bidding list records that correspond to a particular robot, the discount is removed on every frontier point that is close to the robot's currently assigned frontier point (close within h_{rad}). The discount is removed by recalculating information gains, as shown in line 34 of Algorithm 5. Removing the discount is applied only when all robots are busy, so it biases busy robots to keep executing their current command, unless a point with a significantly higher revenue exists.

3.5.5. Calculation of information gain.

In order to calculate the information gain of a frontier point efficiently (without scanning the whole occupancy grid), the information gain of a frontier point in a given occupancy grid is calculated by scanning only the cells that lie within a square surrounding the information gain disk, as shown in Figure 25. Each cell within the square shown in Figure 25 (marked in dashed blue) is checked to see whether it lies within the information gain disk (marked in red). All cells that lie within the information gain disk are counted. The index of the square starting cell is found by first obtaining the index of the frontier point. Next, the offset is subtracted. The offset can be computed knowing the information gain radius and the occupancy grid width and height (the width of an occupancy grid is the number of cells in a row).

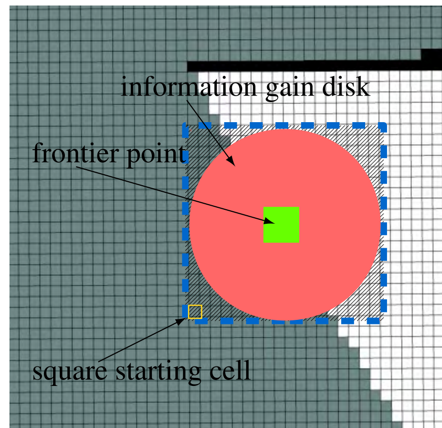


Figure 25: Calculation of information gain for a frontier point

3.6. Implementation

Figure 26 shows an implementation diagram of the exploration strategy. It consists of the SLAM module, path planning module, map merging module, global and local frontier detector modules, the filter module, and the robot task allocator module.

Different ready-made ROS packages are used in the implementation for mapping and path planning. Also, the proposed exploration strategy is itself implemented as a ROS package called "rrt_exploration" [38] consisting of four nodes: the local frontier detector node, the global frontier detector node, the filter node, and the robot task allocator node.

3.6.1. Mapping and localization module (SLAM).

For each robot, this module requires the odometry, the laser scans and the transformation between the laser frame and the robot base frame, in order to provide a local map for the robot and the location of the robot within the map. The ROS "gmapping" package is used for this [31]. It implements a SLAM algorithm that uses a Rao-Blackwellized particle filter [32, 34]. The SLAM algorithm takes robot odometry and laser scans as inputs, and publishes the map as an occupancy grid.

3.6.2. Path planning module.

This module takes the map generated by the SLAM module, robot location, and target assigned point to plan a path and send velocity commands to the robot. For that, the "move_base" node [39], which comes with the ROS "Navigation" stack, is used. This node acts as an interface with several components in the

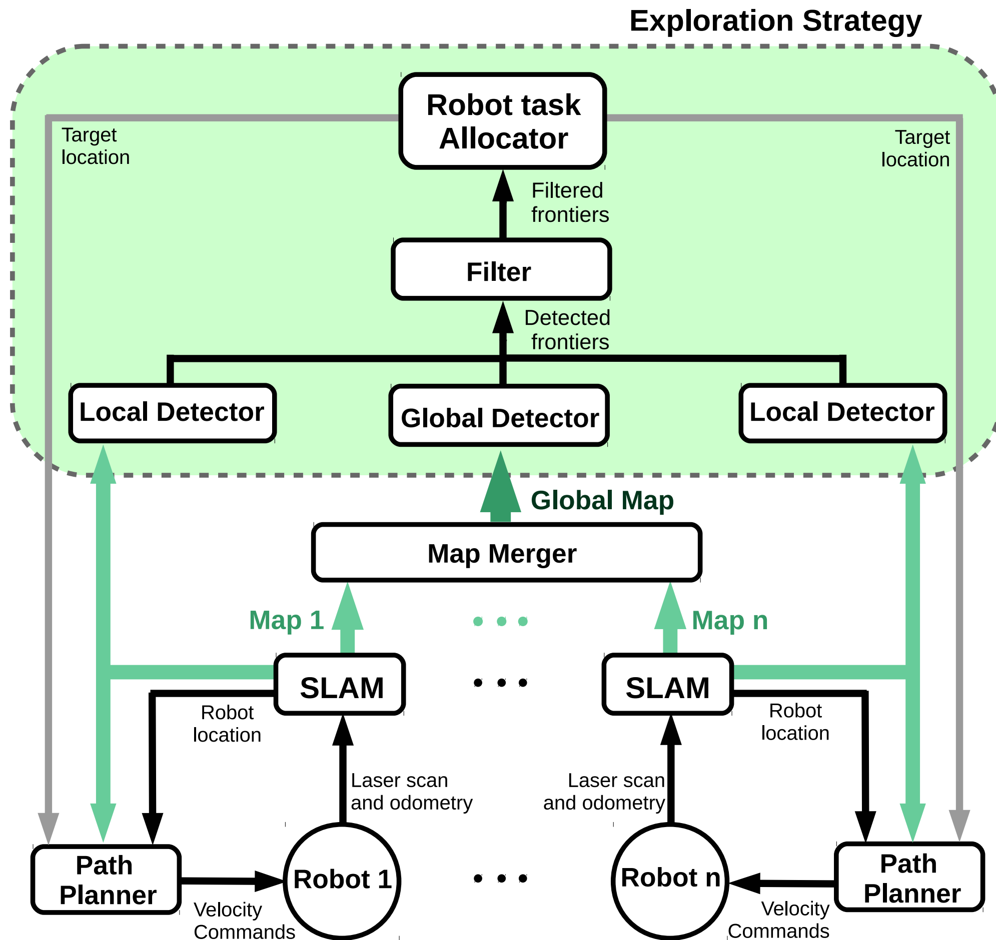
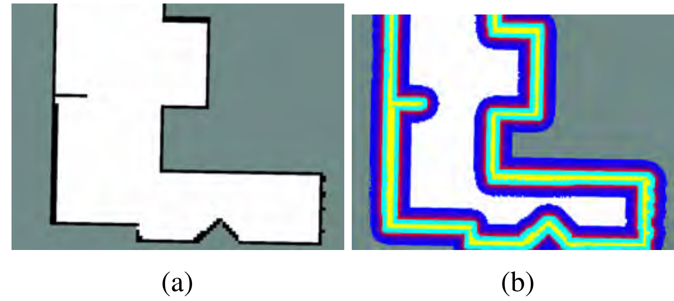


Figure 26: Implementation diagram

"Navigation" stack. It creates local and global costmaps. A costmap is an occupancy gird, but unlike a map, the cell values range from 100 to 0 (no threshold is applied, unlike in a map, where cell values are binary, either 0 or 100). Cell values correspond to costs that are obtained by inflating the obstacle. The closer the cells are to an obstacle, the higher cost values, as shown in Figure 27 (colors correspond to cost values; blue is for low cost value, red is for high cost value, cyan indicates an area where the robot cannot physically reach, and yellow is for the occupied cells). The local costmap is used by the local planner which is responsible for driving the robot to follow the global path created by the global planner, and the global costmap is used by the global planner. Running the "move_base" node also provides robot recovery behaviors that are used when the robot gets stuck. The reader can refer to [40, 41] for more details regarding the local and global planners that are used in the ROS "Navigation" stack. The global planner uses Dijkstra's path planning algorithm (discussed in Section 2.1) for generating the paths.



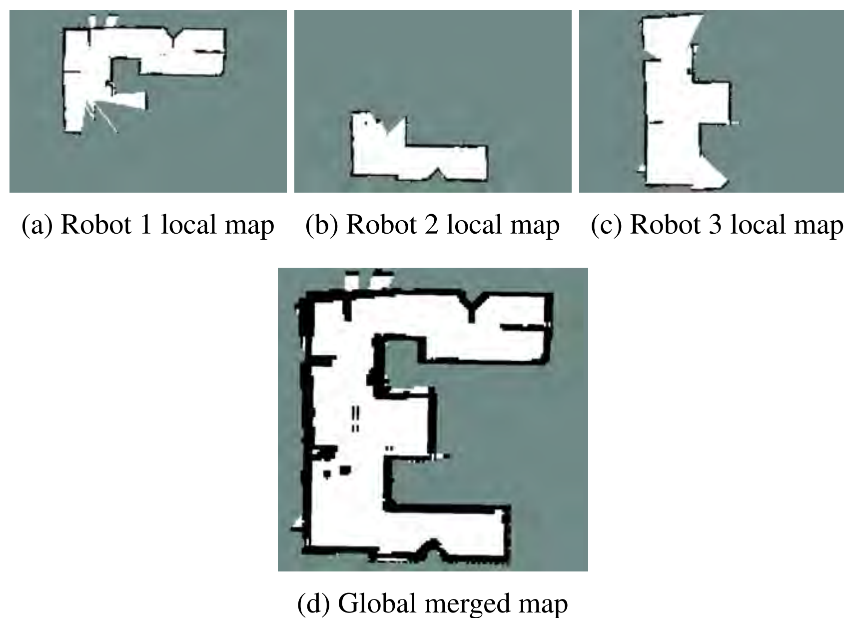
(a) (b)

Figure 27: Costmap showing how obstacles are inflated

An occupancy grid map is shown in (a). In (b) a costmap is drawn over the occupancy grid map, where occupied cells are inflated

3.6.3. Map merging module. This module is required to combine the local maps of all the robots into one global map used by the global detector. Map merging requires frame transformations between local map frames. In our implementation, Robot 1's frame is used as a global frame, and the transformation between other robots' frames and the global frame is assumed to be rigid (i.e. not changing with time).

In order to obtain frame transformation, robot poses are initialized so the relative position and orientation between robots is known. A ready-made ROS package is used for map merging [42]. Figure 28 shows three local maps obtained from the three robots, and the global map obtained after merging them.

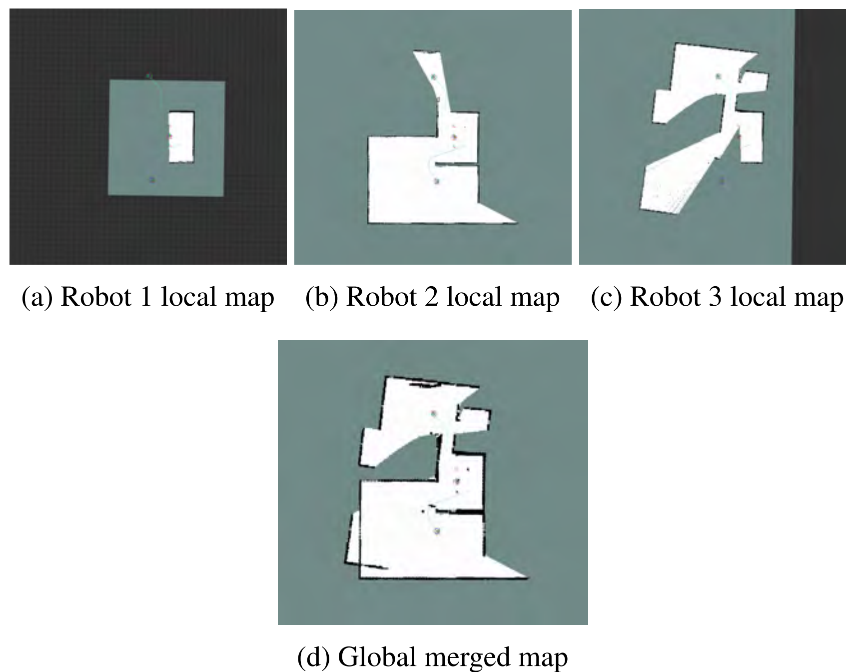


(a) Robot 1 local map (b) Robot 2 local map (c) Robot 3 local map

(d) Global merged map

Figure 28: Map merging example

Map merging has a limitation when it comes to the quality of the merged map when one of the local maps is not accurate. Figure 29 shows an example, where robot's 3 local map was not correct (Robot 3's estimated location exhibited a drift, which affected the process of map building). As a result, the merged map was also affected. Despite this, Robot 3's can still use its local map to navigate without any problems.



(a) Robot 1 local map (b) Robot 2 local map (c) Robot 3 local map

(d) Global merged map

Figure 29: Map merging limitation

3.6.4. Global and local frontier detector modules. The global and local detectors are made as ROS nodes written in C++ to make frontier detection as fast as possible. The node subscribes to the map topic, and publishes the detected frontier points on the "/detected_points" topic, where the message type of this topic is "PointStamped". Normally, in multiple robots configuration, the local frontier detectors take the local maps of the robots, whereas the global frontier detector takes the global merged map.

The node also takes two parameters (i.e. fetches them from the ROS parameter sever); the growth rate (η) of the RRT, and the map topic. Thus, these parameters can be changed according to the user setup (i.e. different setups might have different topic

names). Running multiple instances of the global/local detector causes all the nodes to publish detected points on the same topic the filter node is subscribing to.

3.6.5. Filter module. The filter module is made as a ROS node written in the programming language Python. The filter node subscribes to a ROS topic (default name is "/detected_points"), on which it receives detected frontier points. The message type of the topic is the ROS "PointStamped" message. This message type is used since it carries the point location, and also shows in which coordinate frame the point is represented. Knowing the coordinate frame is needed for deleting invalid frontier points.

The filter node receives detected frontier points from multiple sources (Robot 1, Robot 2, etc..). In order to delete invalid frontier points, the global costmap of each robot is used. When a point is received by the filter node, it reads the coordinate frame associated with the received point. It will then use the global costmap of the robot that corresponds to this coordinate frame for checking the validity of the received point. For example, a frontier point represented in coordinate frame "robot_1/map" will be checked (to see if it is a valid point or not) using the the global costmap of Robot 1. Frontier points that have a cost value above a user-defined threshold in the associated global cost map, will be deleted and considered as invalid. The filter node also calculates the information gain of received frontier points. If the information gain is zero, the frontier point is considered an old point and is therefore deleted.

After filtering the stored frontier points, the filter node publishes all the filtered frontier points on the ROS topic (where the default name is "filtered_points") which the robot task allocator is subscribing to. The message type of this topic is "PointArray", which is a custom defined ROS message that comes with our ROS package [38].

The user can define the following ROS parameters which the filter node accepts:

- "map_topic": Specifies the map topic used by the filter node to delete old frontier points (the map on this topic is used to calculate the information gain of received frontier points). its default value is "/robot_1/map".
- "info_radius": Yhe information gain radius. where the default value is 1 m.
- "costmap_clearing_threshold": The threshold used for clearing invalid frontier points. Any frontier point with a cost value higher than this threshold will be

considered invalid. its default value is 70. Note that higher cost values correspond to cells that are very close to obstacles. These cells are hard or impossible for the robot to reach, which is why such frontier points are considered invalid and are deleted.

- "goals_topic": Topic to receive detected frontier points on. Its default value is "/detected_points".
- "n_robots": Number of robots. Its default value is 1.
- "rate": The rate at which the node runs. Its default value is 100 Hz.

3.6.6. Robot task allocator module. The robot task allocator module is made as a ROS node written in Python. The node by default subscribes to the "/filtered_points" topic on which it receives the filtered frontier points. The node takes the following ROS parameters:

- "map_topic": Specifies the map topic used by the robot task allocator node to calculate the information gain of frontier points. Its default value is "/robot_1/map".
- "info_radius": It is the information gain radius. Its default value is 1 *m*.
- "info_multiplier": This is the weight λ that gives importance to the information gain over the cost of a frontier point, as indicated in Equation 12. Its default value is 3.
- "info_multiplier":
- "hysteresis_radius": The hysteresis radius h_{rad}. Its default value is 3.
- "hysteresis_gain": The hysteresis gain h_{gain}. Its default value is 2.
- "frontiers_topic": The topic name on which the task allocator node will receive the filtered frontier points. Default value is "/filtered_points".
- "n_robots": Number of robots. Its default value is 1.
- "delay_after_assignment": Number of seconds to wait after a robot is assigned a frontier point. Its default value is 0.5 *sec*.
- "rate": The rate at which the node runs. Its default value is 100 Hz.

Algorithm 5 Robot Task Allocator

```
1:  while True do
2:      n_frontiers ← length of Frontiers;
3:      for i = 1,...,n_frontiers do
4:          infoGains[i] ← getInfoGain(Frontiers[i]);
5:      end for
6:      for each robot do
7:          infoGains ← discount(map, infoGains, Frontiers, assigned point);
8:      end for
9:      V ← φ;                                              ▷ Bidding list
10:     R ← φ;                                              ▷ Robots list
11:     C ← φ;                                              ▷ Frontier points list
12:     for each available robot do
13:         for i = 1,...,n_frontiers do
14:             Cost ← norm(Frontiers[i], robot position);
15:             if norm(Frontiers[i], robot position) ≤ h_rad then
16:                 infoGains[i] ← infoGains[i] × h_gain;    ▷ Applying hysteresis gain
17:             end if
18:             Revenue ← λ × infoGains[i] − Cost;
19:             V ← V ∪ Revenue;
20:             R ← R ∪ robot ID;
21:             C ← C ∪ Frontiers[i];
22:         end for
23:     end for
24:     if no available robots then
25:         V ← φ;
26:         R ← φ;
27:         C ← φ;
28:         for each busy robot do
29:             Cost ← norm(Frontiers[i], robot position);
30:             if norm(Frontiers[i], robot position) ≤ h_rad then
31:                 infoGains[i] ← infoGains[i] × h_gain;    ▷ Applying hysteresis gain
32:             end if
33:             if norm(Frontiers[i], assigned point) ≤ h_rad then
34:                 infoGains[i] ← getInfoGain(Frontiers[i]) × h_gain;
35:             end if
36:             Revenue ← λ × infoGains[i] − Cost;
37:             V ← V ∪ Revenue;
38:             R ← R ∪ robot ID;
39:             C ← C ∪ Frontiers[i];
40:         end for
41:     end if
42:     index = Max(V);
43:     Assign(robot with ID = R[index], C[index]);
44: end while
```

Chapter 4: Simulation and Experimental Work

The proposed RRT-based local and global frontier detectors are compared against the image processing-based frontier detector explained in Section 2.4. The same robot task allocator explained in Section 3.5 is used in both cases. Also, the **steer** function used in RRT-based exploration (as shown in line 5 of Algorithms 3 and 4) requires tree growth rate η as an argument. Two simulation maps and one experimental map are presented below, which are explored using our proposed exploration strategy. For each map, we perform a total of 70 exploration runs. Out of these 70, 10 exploration runs are performed using the image processing-based frontier detector. The remaining 60 runs are performed using our proposed local and global frontier detectors. Further, these 60 exploration runs are divided into 6 sets of 10 exploration runs, where the global frontier detector in each set uses a **steer** function with a particular growth rate η, where $\eta \in \{0.5, 1, 4, 6, 10, 15\}$, and the local frontier detectors use a **steer** function with fixed $\eta = 1$. Simulations and experiments are preformed for a single robot, and also for a team of three robots. The total number of exploration runs is 420.

4.1. Simulation Setup

Simulations are carried out using the Gazebo simulator [43], which provides realistic robotic movements, a physics engine, and the generation of sensor data combined with noise.

4.1.1. Robot model. The "kobuki_gazebo" package [44] provides simulation files needed for running the Kobuki robot simulations. After running this package, the simulated Kobuki platform publishes/subscribes to the same topics as a real Kobuki platform. In order to add the lase scanner, the "URDF" [1] files of the Kobuki platform are modified (i.e. by adding the laser scanner plug-in and the laser scanner model link). Figure 30 shows the simulated Kobuki robot with the laser scanner on top.

[1] In Gazebo, "URDF" files describe a robot model in terms of the links the robot consists of, moment of inertia of each link, center of gravity, and even the visual appearance.

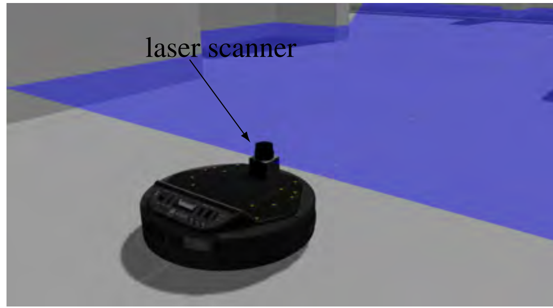


Figure 30: Simulated Kobuki platform

4.1.1.1. Environments used in simulations. Two environments are used for the simulation. The first environment, shown in Figure 31, is a large map with an area of approximately 182 m^2 (free space area), where the robot radius is 0.175 m. In the experiments made using this environment, the laser scanner range is set to 50 m.

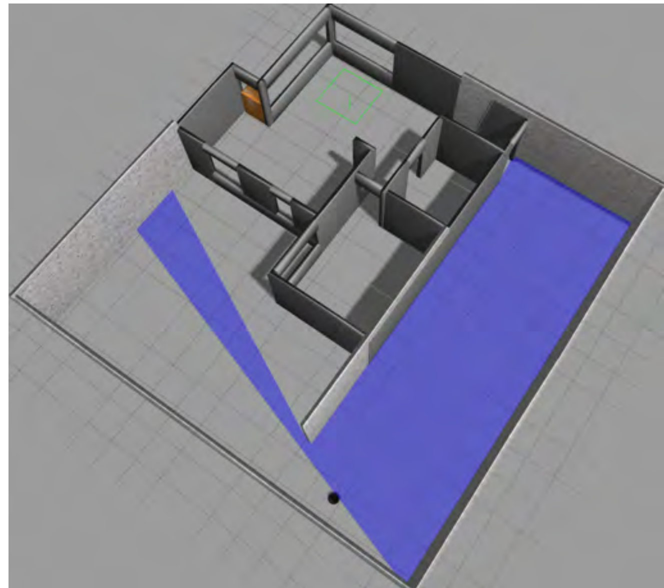


Figure 31: Simulation environment, first map

The second map is made to be very similar to the real map that is actually used in the real setup, and the area of the map is approximately 49 m^2. In the experiments made using this map, the laser scanner range is set to 4 m which is identical to the range of the actual laser scanner used in the real setup. Two sizes of maps and two different laser scanner ranges are used in order to observe the effect of map size and laser scanner range on the proposed exploration strategy.

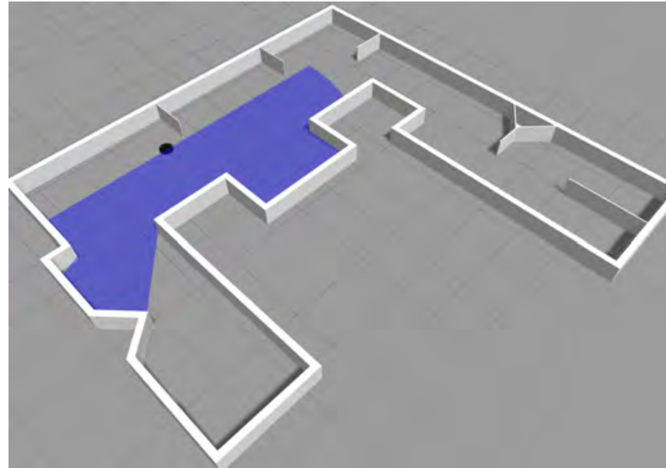


Figure 32: Simulation environment, second map

4.2. Hardware Setup

This section describes the hardware setup used in the experimental work. Figure 33 shows an overview of the hardware components used.

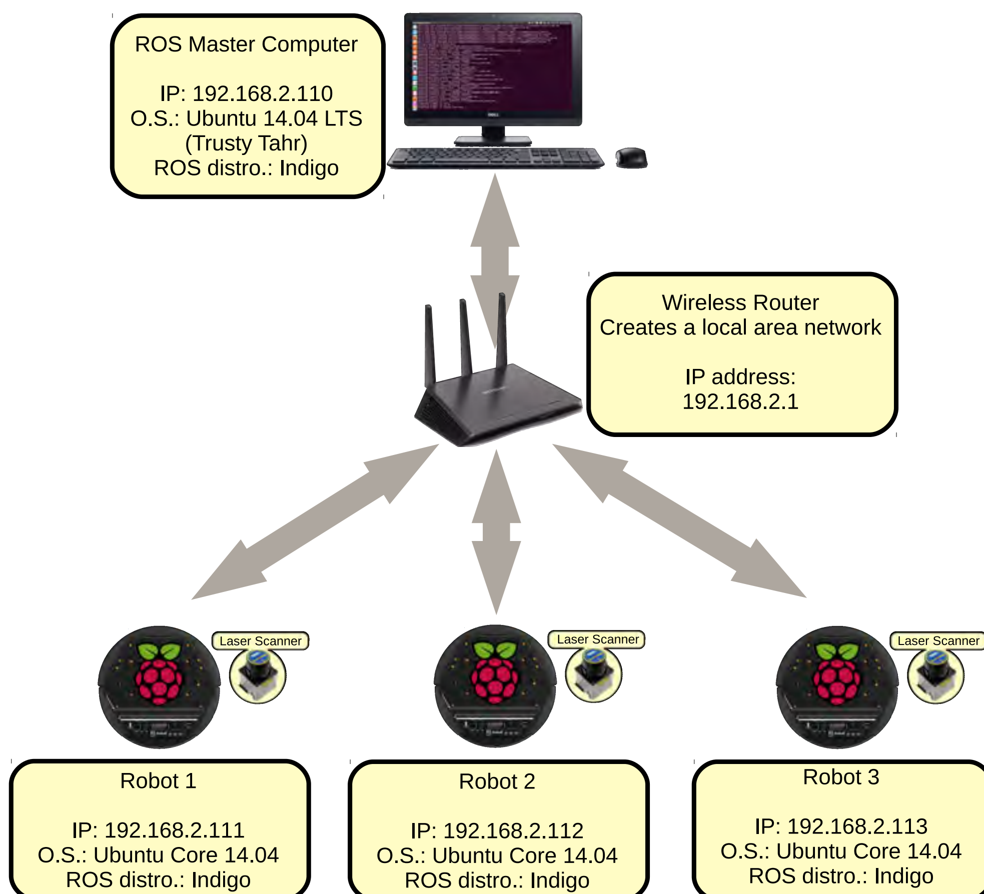


Figure 33: Overview of the hardware setup

4.2.1. Laser scanner. A Hokuyo URG-04LX laser scanner is used. It is a scanning laser range finder used in mapping and navigation, with a scanning area of 240°, and a range of up to 400 *cm*. This laser scanner has a ROS driver that can be used to read the raw data and publish it as a ROS message of type "LaserScan".

In the implementation, the laser scanner is mounted 11.5 *cm* from the robot's center as shown in Figure 34. As a result, a transformation between the laser frame and the robot frame is required, for which a ROS static transformation publisher is used.

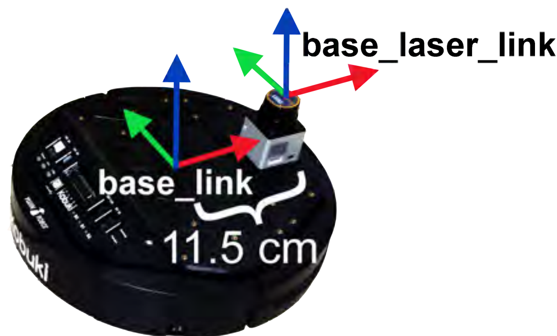


Figure 34: Laser scanner mounting on the Kobuki mobile base

4.2.2. Robot platform. The mobile platform used is the Kobuki robot. It provides odometry measurements, cliff detecting sensors, bumpers, and battery voltage sensing. The Kobuki requires a controller. It supports an embedded microcontroller (where the connection is made through the GPIO pins on the robot), or a computer (where the connection is made through a USB port) that can have either Windows or Linux installed. C++ Kobuki drivers are available for both operating systems.

Since ROS is used in the implementation, each Kobuki robot is connected to a Raspberry Pi (which is a single-board computer, described later) that has Linux Ubuntu installed. The Kobuki robot is supported by ROS and has a package for it. This package provides what is referred to as the Kobuki control system [45]. The Kobuki control system is basically is a collection of nodes that provide additional features such as velocity command smoothening (i.e. preventing jerky motion of the robot), and it also has safety features and velocity command multiplexing. The driver node receives velocity commands on different topics, a node that serves as a multiplexer assigns

different priorities to each topic to avoid any conflict. For example, if the robot hits an obstacle, the bumper sensor will trigger the safety controller node which will send a velocity command to drive the robot backward. If another node sends a velocity command during this process, the multiplexer will give the node a lower priority and prevent it from controlling the robot until the safety control stops sending velocity commands. Without the multiplexer node a conflict will arise. The multiplexer node is called “cmd_vel_mux”.

4.2.3. Robot computer. On each robot, a Raspberry Pi 2 model B is used. The Raspberry Pi 2 is a low-price single-board computer, with 1 GB RAM, and a 900 MHz quad-core CPU. In the implementation, the operating system installed on each Raspberry Pi is Snappy Ubuntu Core 14.04, a special distribution of Ubuntu that provides a minimalistic version of Ubuntu (for instance it does not have a graphical user interface), but supports ROS. The chosen ROS distribution is Indigo. With the low specifications of the Raspberry Pi, the computation on it is limited to only run the Kobuki driver node and the laser scanner node, where the master computer does the rest of the computation. Figure 35 shows the three robots used in the experiments.

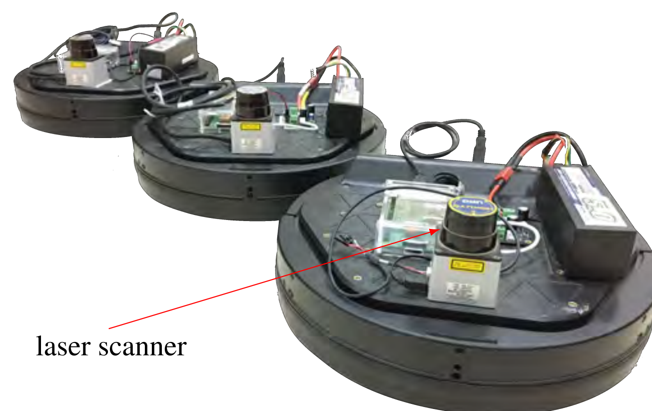


Figure 35: Robots used in the experiments

4.2.4. Master computer. The master computer has Ubuntu 14.04 and ROS Indigo installed. It runs the “gmapping” SLAM node and the “move_base” node for all the robots. It also runs the map merging node, the local and global detector nodes, the

filter node, and the robot task allocator node. The master computer is connected to the robots using a WiFi connection through a router. The master computer has an i7-2600 CPU (quad-core, 3.40 GHz), 4 GB RAM, and an AMD Radeon HD 6350 GPU.

4.2.5. Network setup. A ROS network can be setup by running the ROS master on one machine (by running the "`roscore`" command on the master computer). This will create a ROS network but on the local machine only. For other robots to connect to the ROS network, the "`ROS_MASTER_URI`" environment variable has to be set to point to the master computer. In our setup, this is done by adding the following line to the "`.bashrc`" file. The "`.bashrc`" file is a bash script that is run every time you open a shell or a terminal window. It is located in the home directory (i.e. "`~/.bashrc`" is the path for this file). It's file name starts with a dot indicating that it is a hidden file.

```
# This line is added to the end of the .bashrc file in each robot
# "hassandesktop" is the master hostname

export ROS_MASTER_URI=http://hassanNewDesktop:11311
```

In addition, each machine on the network should have its host name added to the "`hosts`" file of all the other machines. The "`hosts`" file path is "`/etc/hosts`". For example, the master machine "`hosts`" file has the following:

```
127.0.0.1       localhost
127.0.1.1       hassanNewDesktop

# The following lines are desirable for IPv6 capable hosts

::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

192.168.2.111 robot1
192.168.2.112 robot2
192.168.2.113 robot3
```

In a ROS network, the clocks of all the machines have to be in sync (many ROS messages are time stamped, and a large time offset between clocks can cause errors). For that, the "`chrony`" tool is used by setting the master computer as a server (i.e. the master computer will be the clock source, to which all the robots will sync their clocks). This is not always required; however, in our setup it is needed.

4.2.6. The map used in the experiment. The map where the real experiments are conducted is shown in Figure 36. The area of this map is approximately 49 m^2.

Figure 36: The real map used in the experiments

4.3. Results

As described at the beginning of this chapter, 70 exploration runs are performed for 3 maps (2 simulation, 1 real). The outcome of each exploration experiment is an occupancy gird. Figure 37 shows the occupancy grids obtained for the simulated environments, and the real map using a single robot. At the end of each exploration run, the total time taken for exploration and the total distance covered by the robot during exploration are recorded.

4.3.1. Simulation results. The first set of results is for the first environment with the large map, shown in Figure 31, where the long-range laser scanner is used. The results for this part are shown in Figure 39.

(a) (b) (c)

Figure 37: Occupancy grid maps generated using the proposed exploration strategy, single robot case.

Picture (a) is the occupancy grid of the first simulation environment, (b) is the occupancy grid of the second simulation environment, and (c) is the occupancy grid of the real map, showing the robot after it has finished exploration.

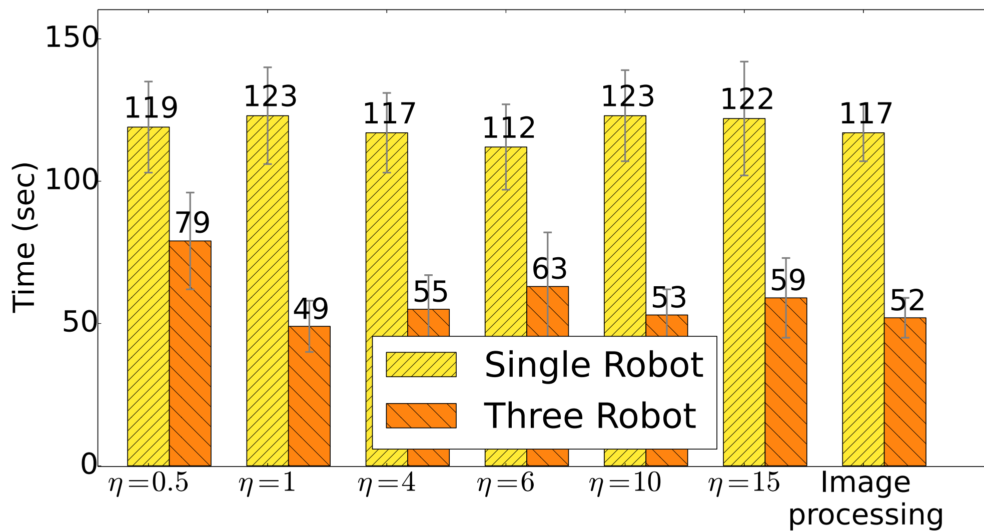
(a) (b) (c)

Figure 38: Occupancy grid maps generated using the proposed exploration strategy, three robots case.

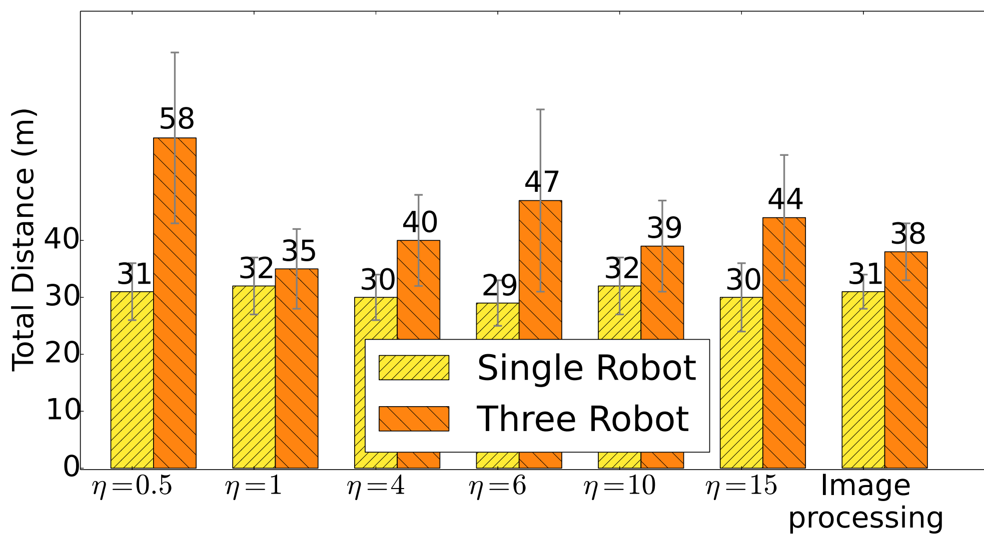
Picture (a) is the occupancy grid of the first simulation environment, (b) is the occupancy grid of the second simulation environment, and (c) is the occupancy grid of the real map. All these maps are obtained by merging the local maps of all robots

The second set of simulation results is for the second environment with a small map, shown in Figure 32, where a low-range laser scanner is used. The results for this part are shown in Figure 40.

The simulation results show that using RRT-based detection does not compromise the efficiency of exploration in terms of the time and total distance needed to cover the map. The effect of the laser scanner range is also insignificant, although it affects the speed at which RRT expands in the space. The effect of the laser scanner range is



(a)

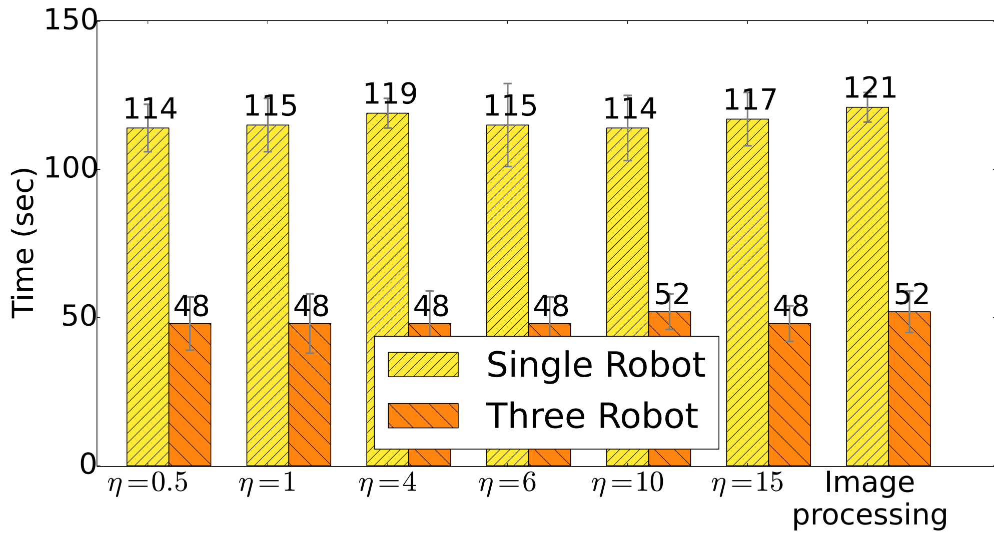


(b)

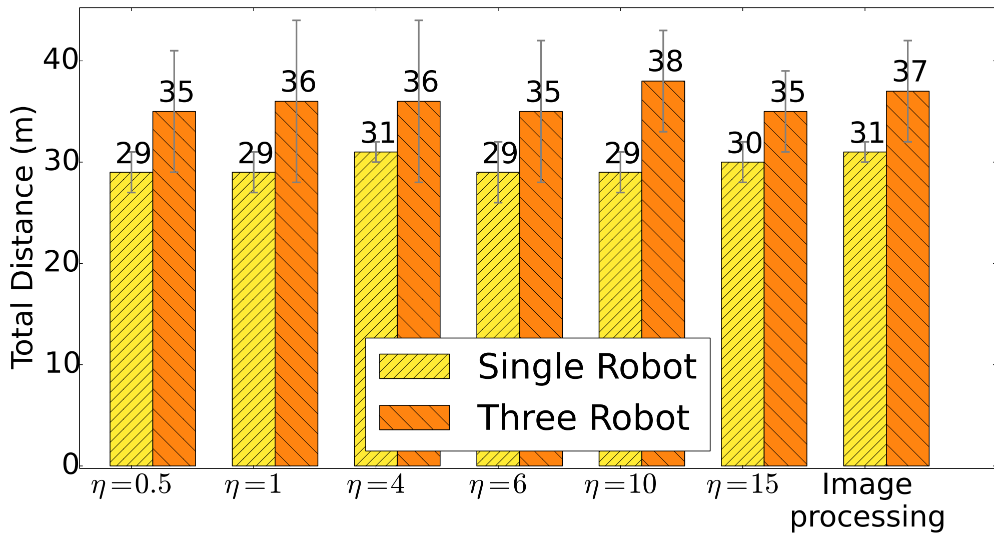
Figure 39: Large map simulation results

compensated by the multiple RRTs (a global frontier detector which never resets, and a local frontier detector which resets frequently).

The simulation results for the single robot and for the first (large) environment show that the time needed to finish exploration using the proposed algorithm is less than 123 seconds on average. In contrast with an earlier work of the proposed algorithm, which is similar to the work in [7,46], where the robot is made to follow the edges of RRT as the tree grows, the obtained exploration time for the same simulation environment



(a)



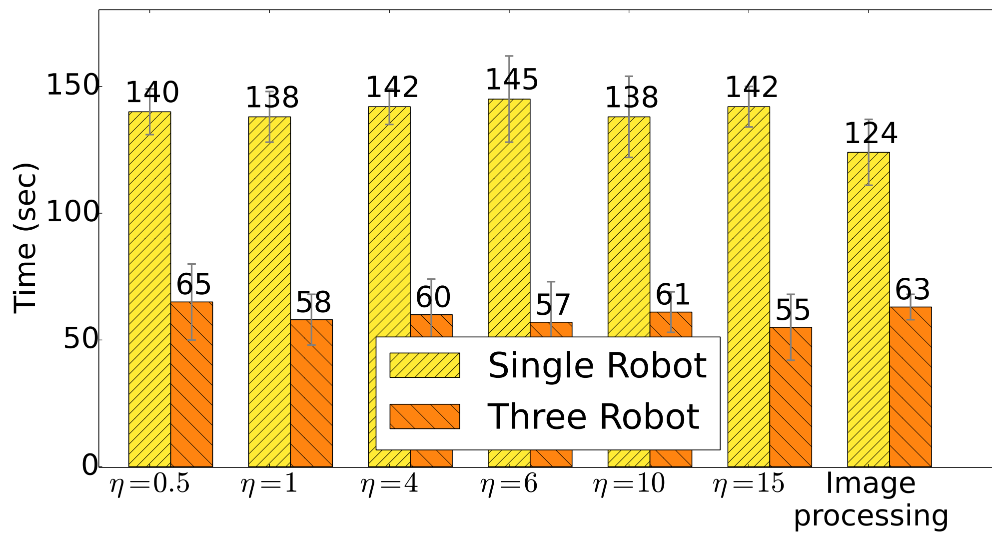
(b)

Figure 40: Small map simulation results

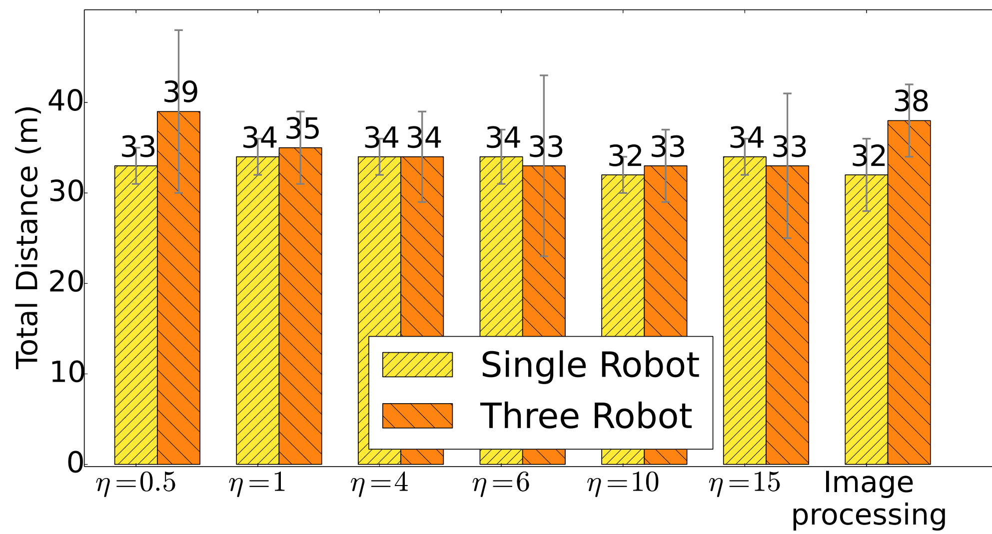
was more than 300 seconds. This shows that not following tree edges as the RRT grows results in a substantial improvement in exploration speed.

4.3.2. Experimental results. The last set of results is produced using the real experimental setup. Figure 41 shows the experimental results. The results agree in general with the above mentioned simulation results. The only difference is that the RRT-based experimental exploration is slightly more time consuming than the image processing-based exploration. However we believe that such small differences

in performance show that the proposed exploration strategy will be viable for 3-D exploration, where image processing-based exploration techniques may be unusable.



(a)



(b)

Figure 41: Real map results

Chapter 5: Conclusion and Future Work

5.1. Conclusion

In this work, a new map exploration strategy is presented. The strategy is based on the Rapidly-exploring Random Tree (RRT) algorithm, where RRT is used to find frontier regions. Usual implementations of frontier detectors utilize image processing tools to extract frontier regions, limiting their application to 2-D exploration. The proposed strategy uses RRT to detect frontier points. RRT is not limited to 2-D space, hence, it can be applied to find frontier points in 3-D maps, which can in the future allow for efficient 3-D exploration.

The proposed exploration strategy is implemented using the Robot Operating System (ROS). It is tested using simulations and real experiments, with a total number of 420 exploration runs. Simulations are carried out using the Gazebo simulator. Whereas the real experiments are conducted using three mobile robots equipped with laser scanners. The map representation that is used in the experiments is the 2-D occupancy gird map. Two simulation environments of different sizes are used in the simulation in order to test the effect of map size on the speed of exploration. For each exploration run two metrics are recorded; the total time needed to finish exploring the map, and the cost which is defined as the total distance traveled by all the robots. Additionally, different values of the RRT growth rate η are tested. The proposed strategy is also compared against an image processing-based frontier detection algorithm.

The results show that the proposed strategy can successfully extract frontiers and explore the entire map in a reasonable amount of time and cost, with minimal sacrifices to performance when compared against the image processing-based frontier detection algorithm. Another contribution of this work is that a custom ROS package for RRT-based exploration has been developed, and it is available for users at [38].

5.2. Future Work

This work shows that RRT-based frontier detection can be used for detecting frontier points in 2-D maps having similar performance compared to existing algorithms.

However, the main advantage of using RRT for frontier point detection is that it can be used on 3-D maps, where other existing algorithms might fail. The proposed RRT-based frontier detection algorithm can be extended to 3-D map representations such as OctoMaps [47].

Another improvement that can be added is the distribution of computation. Due to the low performance of the Raspberry Pi computer used on each robot, all the processing (SLAM, path planning, running the RRT-based frontier point detectors, and the robot task allocator) was made on the master computer. Instead, each robot can run the followings locally; i) SLAM module, ii) path planning module, and iii) the local RRT-based frontier detector module. In order to be able to do that, a more powerful computer has to be used on each robot. This reduces the data sent over the network which helps in speeding up exploration.

Additionally, the exploration strategy can be changed from being centralized to being hybrid. The modules which make the proposed exploration strategy a centralized approach are the robot task allocator module and the filter module; all other modules can be run locally on each robot. Thus, each robot can do exploration on its own using a local task allocator module. Every time a robot comes in the range of another robot, one of them becomes a master and starts running the global task allocator commanding the two robots. This requires the use of an **ad hoc** network.

More work can also be added to the exploration strategy itself. The current robot task allocator does not take into account map quality (the probability values of each cell in the occupancy grid map, i.e. $p(m_i)$), as the SLAM module runs independently from other modules. This can be changed so the robot task allocator takes into account map quality and robot pose estimates, so it generates target points that increase map quality in addition to exploring new regions at the lowest cost. Also, more work can be added to fix the problem of map merging. The current solution merges the local maps given that the initial poses of the robots are known. If one of the local maps is not accurate, the obtained global map is also affected and it becomes unusable for navigation.

Finally, the exploration strategy was tested in small environments. Large environments require long-range laser scanners (a range above 20 m).

References

[1] B. Yamauchi, "A frontier-based approach for autonomous exploration," in *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA '97)*, Jul 1997, pp. 146–151.

[2] B. Yamauchi, "Frontier-based exploration using multiple robots," in *Proceedings of the Second International Conference on Autonomous Agents (AGENTS '98)*. New York, NY, USA: ACM, 1998, pp. 47–53.

[3] M. Keidar and G. A. Kaminka, "Robot exploration with fast frontier detection: Theory and experiments," in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems (AAMAS), 2012, pp. 113–120.

[4] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep., 1998.

[5] Y. Wang, A. Liang, and H. Guan, "Frontier-based multi-robot map exploration using particle swarm optimization," in *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, April 2011, pp. 1–6.

[6] P. G. C. N. Senarathne *et al.*, "Efficient frontier detection and management for robot exploration," in *Cyber Technology in Automation, Control and Intelligent Systems (CYBER), 2013 IEEE 3rd Annual International Conference on*, May 2013, pp. 114–119.

[7] G. Oriolo *et al.*, "The SRT method: randomized strategies for exploration," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '04)*, vol. 5, April 2004, pp. 4688–4694.

[8] H. El-Hussieny, S. F. M. Assal, and M. Abdellatif, "Improved backtracking algorithm for efficient sensor-based random tree exploration," in *The Fifth International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN)*, June 2013, pp. 19–24.

[9] A. Franchi *et al.*, "The sensor-based random graph method for cooperative robot exploration," *IEEE/ASME Transactions on Mechatronics*, vol. 14, no. 2, pp. 163–175, April 2009.

[10] A. Franchi *et al.*, "A randomized strategy for cooperative robot exploration," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, April 2007, pp. 768–774.

[11] A. Franchi *et al.*, "A decentralized strategy for cooperative robot exploration," in *Proceedings of the 1st International Conference on Robot Communication and Coordination (RoboComm '07)*. Piscataway, NJ, USA: IEEE Press, 2007, pp. 7:1–7:8.

[12] C. Stachniss, G. Grisetti, and W. Burgard, "Information gain-based exploration using rao-blackwellized particle filters," in *Proceedings of Robotics: Science and Systems (RSS)*, 2005, pp. 65–72.

[13] F. Bourgault *et al.*, "Information based adaptive robotic exploration," in *The IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 1, 2002, pp. 540–545.

[14] R. Zlot *et al.*, "Multi-robot exploration controlled by a market economy," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '02)*, vol. 3, 2002, pp. 3016–3023.

[15] Z. Yan *et al.*, "Team size optimization for multi-robot exploration," in *Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2014)*, Bergamo, Italy, October 2014, pp. 438–449.

[16] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[17] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[18] S. M. LaValle, *Planning Algorithms*. New York, NY, USA: Cambridge University Press, 2006.

[19] A. Romero. (2014, Jun.) Ros concepts. Internet. [Online]. Available: http://wiki.ros.org/ROS/Concepts [Accessed: Mar 12, 2016].

[20] T. Foote and M. Purvis. (2014, Dec.) Standard units of measure and coordinate conventions. Internet. [Online]. Available: http://www.ros.org/reps/rep-0103.html [Accessed: Apr 13, 2016].

[21] I. Saito. (2015, Jul.) Ros tf package. Internet. [Online]. Available: http://wiki.ros.org/tf [Accessed: Apr 13, 2016].

[22] W. Meeussen. (2010, Oct.) Coordinate frames for mobile platforms. Internet. [Online]. Available: http://www.ros.org/reps/rep-0105.html [Accessed: Apr 13, 2016].

[23] S. Kohlbrecher. (2012, Aug.) Hector slam tutorials. Internet. [Online]. Available: http://wiki.ros.org/hector_slam/Tutorials/SettingUpForYourRobot [Accessed: Apr 13, 2016].

[24] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[25] S. Thrun, "Learning metric-topological maps for indoor mobile robot navigation," *Artificial Intelligence*, vol. 99, no. 1, pp. 21–71, 1998.

[26] Occupancygrid message. Internet. [Online]. Available: http://docs.ros.org/kinetic/api/nav_msgs/html/msg/OccupancyGrid.html [Accessed: Apr 10, 2017].

[27] C. Zhu *et al.*, "A 3d frontier-based exploration tool for MAVs," in *The 2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, Nov 2015, pp. 348–352.

[28] C. Dornhege and A. Kleiner, "A frontier-void-based approach for autonomous exploration in 3d," in *The 2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*, Nov 2011, pp. 351–356.

[29] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[30] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 790–799, Aug 1995.

[31] S. Schweigert. (2015, Dec.) Ros gmapping. Internet. [Online]. Available: http://wiki.ros.org/gmapping [Accessed: Feb 2, 2017].

[32] G. Grisettiyz, C. Stachniss, and W. Burgard, "Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, April 2005, pp. 2432–2437.

[33] D. Comaniciu and P. Meer, "Mean shift: a robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 5, pp. 603–619, May 2002.

[34] G. Grisetti, C. Stachniss, and W. Burgard, "Improved techniques for grid mapping with rao-blackwellized particle filters," *IEEE Transactions on Robotics*, vol. 23, no. 1, pp. 34–46, Feb 2007.

[35] S. Mukhopadhyay and F. Zhang, "A path planning approach to compute the smallest robust forward invariant sets," in *Proceedings of the American Control Conference*, June 2014, pp. 1845–1850.

[36] P. Varnell, S. Mukhopadhyay, and F. Zhang, "Discretized boundary methods for computing smallest forward invariant sets," in *Proceedings of the 55th IEEE Conference on Decision and Control (CDC)*, Dec 2016, pp. 6518–6524.

[37] R. Simmons *et al.*, "Coordination for multi-robot exploration and mapping," in *Proceedings of the AAAI National Conference on Artificial Intelligence*. Austin, TX: AAAI, 2000.

[38] H. Umari. (2016, Sep.) RRT exploration ROS package. Internet. [Online]. Available: https://github.com/hasauino/rrt_exploration [Accessed: Mar 1, 2017].

[39] D. Lu. (2016, Mar.) Move base package summery. Internet. [Online]. Available: http://wiki.ros.org/move_base [Accessed: Apr 28, 2016].

[40] S. Maniatopoulos. (2016, Feb.) Ros global planner. Internet. [Online]. Available: http://wiki.ros.org/global_planner [Accessed: Feb 5, 2017].

[41] I. Gavran. (2017, Feb.) Ros local planner. Internet. [Online]. Available: http://wiki.ros.org/base_local_planner [Accessed: Apr 12, 2017].

[42] J. Horner. (2016, May) Multirobot map merge package. Internet. [Online]. Available: http://wiki.ros.org/multirobot_map_merge [Accessed: Mar 25, 2017].

[43] Gazebo simulator. Internet. [Online]. Available: http://gazebosim.org/ [Accessed: Mar 1, 2017].

[44] M. Liebhardt. (2013, Sep.) Ros "kobuki_gazebo package. Internet. [Online]. Available: http://wiki.ros.org/kobuki_gazebo [Accessed: Feb 5, 2017].

[45] J. S. Simon. (2013, Jul.) Kobuki's control system. Internet. [Online]. Available: http://wiki.ros.org/kobuki/Tutorials [Accessed: Apr 14, 2016].

[46] A. Bircher *et al.*, "Receding horizon "next-best-view" planner for 3d exploration," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 1462–1468.

[47] A. Hornung *et al.*, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, 2013, software available at http://octomap.github.com.

Appendix A: Raw Data Results

Single Robot, Simulation, First (Large) Map Results

Exp. No.	Global RRT Detector												Image processing	
	Eta=1		Eta=4		Eta=0.5		Eta=6		Eta=15		Eta=10			
	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)
1	138	36.49	97	22.5	117.18	31.2	123	31.09	104	23.53	137	37.45	121	32.97
2	118	33.03	101	26.11	106	26.67	121	29.9	138	35.82	128	33.55	126	35.59
3	101	23.7	117	31.13	143	35.86	129	32.76	153	38.19	133	31.51	132	31.41
4	154	40.93	126	31.54	98	27.16	99	25.25	135	34.64	132	34.69	126	33.85
5	113	31.43	113	26.7	110	27.36	103	26.43	97	25.1	98	24.21	124	31.51
6	119	28.43	126	33.92	107	26.25	108	30.19	116	32.3	111	31.1	104	26.71
7	108	25.84	114	31.74	125	31.71	102	28.17	118	33.32	122	33.56	111	30.65
8	143	37.48	134	31.05	120	33.62	107	28.71	144	35.4	141	37.89	107	29.09
9	117	33.43	104	27.65	147	41	90	23.75	92	23.13	96	21.79	116	33.26
10	119	31.84	136	37.26	113	30	138	37.16	123.34	23.34	134	34.9	107	28
AVG.	**123**	**32.26**	**116.8**	**29.96**	**118.618**	**31.083**	**112**	**29.341**	**122.034**	**30.477**	**123.2**	**32.065**	**117.4**	**31.304**

Figure 42: Raw data results for single robot, simulation, first (large) map

Single Robot, Simulation, Second (Small) Map Results

Exp. No.	Global RRT Detector												Image processing	
	Eta=1		Eta=4		Eta=0.5		Eta=6		Eta=15		Eta=10			
	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)
1	118	30	119	31.1	110	29.51	120	30.62	108	27.98	117	29.82	126	31.75
2	124	31.46	119	30.38	120	31.62	120	30.35	126	32.17	86	22.73	120	31.08
3	110	28.98	112	29.38	120	29.23	114	27.91	117.58	29.42	113	28.82	120	31.51
4	107	27.8	109	28.15	117	30.38	134	33.02	117	30.07	110	28.28	114	29.95
5	121	29.77	118	30.02	126	32.37	107	27.9	99	25.22	116	29.35	126	31.79
6	126.56	32.19	124	31.96	109	29.07	96	24.59	123	31.66	116	29.84	126	31.34
7	115	28.81	123.07	32.15	108	29.05	125	31.76	128	32.07	113	30.07	126	31.7
8	110	28.25	115	29.78	113	28.37	88	23.16	115	30.13	122	30.24	125	32.27
9	97	24.21	122	31.69	120	30.51	114	28.36	121	29.53	127	29.89	112	29.91
10	117	31.62	126	31.39	97	24.58	129	30.2	119	29.16	121	30.01	117	30.95
AVG.	**114.556**	**29.309**	**118.707**	**30.6**	**114**	**29.469**	**114.7**	**28.787**	**117.358**	**29.741**	**114.1**	**28.905**	**121.2**	**31.225**

Figure 43: Raw data results for single robot, simulation, second (small) map

Single Robot Real Experimental Results

Exp. No.	Global RRT Detector												Image processing	
	Eta=1		Eta=4		Eta=0.5		Eta=6		Eta=15		Eta=10			
	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)	Time (sec)	Dist. (m)
1	138	34.12	144	32.96	140	33.6	153	34.22	146	32.89	133	31.33	136	32.65
2	131	33	136	32.24	144	33.21	133	32.68	146	32.05	146	32.03	120	31.38
3	136	31.42	138	34.42	150	32.35	152	34.82	139	35.42	127	32.05	106	27.65
4	127	31.1	141	32.26	138	33.87	150	34.39	128	31.41	147	33.94	118	30.85
5	134	32.21	141	33.5	127	27.78	136	32.97	148	35.04	123	32.42	144	35.38
6	126.86	32.06	142	34.67	158	34.53	157	34.16	152	36.48	142	32.4	124	32.25
7	149	36.03	142	32.23	139	32.98	181	41.61	145	34.29	120	32.35	143	39.26
8	140	36.23	135	34.12	137	31.26	142	32.75	133	33.37	121	25.73	115	30.71
9	145	35.67	143	31.9	135	30.93	120	32.47	146	35.06	152	34.94	112	26.67
10	157	36.04	159	38.31	136	34.8	128	32.78	136	30.69	168	33.47	121	32.32
AVG.	**138.386**	**33.788**	**142.1**	**33.661**	**140.4**	**32.531**	**145.2**	**34.285**	**141.9**	**33.67**	**137.9**	**32.066**	**123.9**	**31.912**

Figure 44: Raw data results for single robot real experiments

Three Robots, Simulation, First (Large) Map Results

Exp. No.	RRT Detector															Image based				
	Eta=1					Eta=4					Eta=0.5									
	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)
1	48	10.02	12.84	11.38	34.24	54	12.89	13.97	13.01	39.87	73	17.59	19.41	18.6	55.6	51	13.7	11.74	11.83	37.27
2	69.79	16.27	18.14	17.36	51.77	69	16.18	19.12	12.49	47.79	84	19.29	23.69	19.47	62.45	63	16.23	15.05	13.17	44.45
3	53	13.41	13.29	13.19	39.89	39	9.44	9.93	10.18	29.55	93	23.79	25.2	25.5	74.49	46	12.06	11.82	11.67	35.55
4	42	9	11.06	9.21	29.27	72	18.18	18.65	13.94	50.77	93	18.97	25.08	23.38	67.43	49	13.59	11.93	12.1	37.62
5	53	7.69	13.13	13.03	33.85	55	13.62	15.05	12.95	41.62	74	13.68	18.07	17.79	49.54	44	11.01	11.02	10.52	32.55
6	43.49	11.06	11.65	11.95	34.66	56	12.23	14.7	15.38	42.31	51	5.82	12.66	12.11	30.59	45	10	10.71	10.3	31.01
7	50	12.25	12.3	12.1	36.65	40	8.25	11.8	9.3	29.35	66	14.87	16.54	15.13	46.54	51	12.14	13.01	11.19	36.34
8	38.69	9.44	9.3	9.92	28.66	41.55	11.9	10.3	8.31	30.51	93	24.18	23.49	23.92	71.59	50	13.02	11.79	12.83	37.64
9	41	10.83	11.13	9.27	31.23	65	12.62	14.4	17.35	44.37	61	16.04	16.5	12.48	45.02	65	16.23	17.33	13.53	47.09
10	49	12.33	12.94	9.38	34.65	58	14.09	15.17	14.53	43.79	105	26.52	26.44	21.9	74.86	54	14.43	11.79	14.62	40.84
AVG	**48.797**	**11.23**	**12.578**	**11.679**	**35.487**	**54.955**	**12.94**	**14.309**	**12.744**	**39.993**	**79.3**	**18.075**	**20.708**	**19.028**	**57.811**	**51.8**	**13.241**	**12.619**	**12.176**	**38.036**

Exp. No.	Eta=6					Eta=15					Eta=10				
	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)
1	50	10.05	11.97	12.58	34.6	67	17.38	18	16.96	52.34	49	10.47	12.05	11.89	34.41
2	73	19.02	19.45	20.06	58.53	63	14.04	16.99	16.19	47.22	62	15.93	16.47	15.39	47.79
3	65	15.42	17.18	12.57	45.17	37	9.64	9.57	9.08	28.29	44	9.06	11.53	10.5	31.09
4	113	29.34	27.09	29.81	86.24	58	14.98	12.96	14.31	42.25	52	12.98	12.96	12.32	38.26
5	46.4	11.08	11.93	12.46	35.47	85	21.92	22.98	21.06	65.96	51	11.25	13.05	12.83	37.13
6	54	13.31	13.86	9.4	36.57	67	15.93	18.36	14.13	48.42	64	15.64	16.76	17.07	49.47
7	57	13.98	14.38	13.02	41.38	67	17.65	15.28	13.11	46.04	63	16.06	17.73	11.81	45.6
8	68	17.19	17.36	18.2	52.75	42	10.21	11.47	10.72	32.4	44	10.6	10.86	12.57	34.03
9	57	13.09	15.6	14.7	43.39	53	12.9	13.5	14.56	40.96	41	9.55	8.36	10.65	28.56
10	49	11.29	9.41	13.54	34.24	53	12.97	10.51	13.18	36.66	64	15.15	17.23	14.71	47.09
AVG	**63.24**	**15.377**	**15.823**	**15.634**	**46.834**	**59.2**	**14.762**	**14.962**	**14.33**	**44.054**	**53.4**	**12.669**	**13.7**	**12.974**	**39.343**

Figure 45: Raw data results for three robots, simulation, first (large) map

Three Robots, Simulation, Second (Small) Map Results

Exp. No.	RRT Detector															Image based				
	Eta=1					Eta=4					Eta=0.5									
	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)
1	43	10.4	10.55	8.27	29.22	42	10.65	10.8	11.68	33.13	35	9.51	9.33	8.34	27.18	56	14.29	12.37	12.64	39.3
2	42	9.58	8.55	10.14	28.27	51	11.84	12.52	10.71	35.07	40	10.45	7.04	10.47	27.96	52	13.54	11.67	12.17	37.38
3	43	11.48	10.58	11.32	33.38	76	19.27	18.72	20.3	58.29	51	13.86	13.27	12.39	39.52	49	12.44	12.34	11.86	36.64
4	42	11.27	9.14	8	28.41	49	11.84	11.6	12.57	36.01	42	9.86	10.12	8.99	28.97	50	13.43	11.25	11.28	35.96
5	61	14.58	16.56	16.52	47.66	39	10.72	9.62	10.19	30.53	63	15.09	15.19	15.08	45.36	49	13.19	10.21	11.61	35.01
6	41	11.8	9.94	11.21	32.95	49	11.7	10.68	11.09	33.47	49	12.85	10.53	12.49	35.87	46	12.16	10.66	9.74	32.56
7	66	16.39	17.76	17.02	51.17	42	11.58	10.76	10.88	33.22	47	10.81	12.48	10.76	34.05	52	13.41	10.23	11.32	34.96
8	39	11.24	9.72	11.1	32.06	39	9.42	9.07	10.25	28.74	63	12.83	15.83	15.15	43.81	55	13.31	10.48	13.11	36.9
9	46	12.34	11.48	11.03	34.85	46	11.03	11.77	13.01	35.81	45	12.85	11.85	11.91	36.61	70	15.22	17.01	17.16	49.39
10	56	13.9	13.77	13.49	41.16	49	12.2	11.38	13.49	37.07	42	11.13	11.17	8.48	30.78	45	11.72	9.91	10.98	32.61
AVG	**47.9**	**12.298**	**11.805**	**11.81**	**35.913**	**48.2**	**12.025**	**11.692**	**12.417**	**36.134**	**47.7**	**11.924**	**11.681**	**11.406**	**35.011**	**52.4**	**13.271**	**11.613**	**12.187**	**37.071**

Exp. No.	Eta=6					Eta=15					Eta=10				
	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)
1	61	14.43	15.38	15.88	45.69	51	11.68	11.98	13.14	36.8	44	11.49	10.87	10.48	32.84
2	59	13.87	13.97	15.08	42.92	48	12.43	10.18	11.77	34.38	46	11.43	10.11	9.86	31.4
3	38	9.78	6.83	8.3	24.91	53	12.07	13.7	12.95	38.72	59	13.64	13.72	14.78	42.14
4	55	11.96	12.51	14.34	38.81	43	10.7	9.1	11.5	31.3	60	13.19	16.37	16.14	45.7
5	54	13.54	13.29	12.72	39.55	38	9.7	9.34	8.8	27.84	48	9.11	11.72	13.38	34.21
6	44	10.91	11.23	11.15	33.29	57	13.73	14.11	14.82	42.66	58	13.2	15.2	12.86	41.26
7	38	10	7.98	8.42	26.4	53	13.85	13.85	8.59	36.29	46	11.5	11.29	11.54	34.33
8	54	12.86	14.44	12.92	40.22	44	10.73	10.94	11.89	33.56	50	9.61	12.22	13.21	35.04
9	43	10	10.75	10.2	30.95	43	9.74	11.88	11.56	33.18	54	12.94	13.44	14.56	40.94
10	38	9.44	9.96	7.16	26.56	45	11.45	11.62	11.79	34.86	50	12.41	11.89	13.35	37.65
AVG	**48.4**	**11.679**	**11.634**	**11.617**	**34.93**	**47.5**	**11.608**	**11.67**	**11.681**	**34.959**	**51.5**	**11.852**	**12.683**	**13.016**	**37.551**

Figure 46: Raw data results for three robots, simulation, second (small) map

Three Robots Real Experimental Results

Exp. No.	RRT Detector															Image based				
	Eta=1					Eta=4					Eta=0.5									
	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)
1	72	16.52	11.22	7.69	35.43	58	11.55	11.13	11.08	33.76	62	11.02	10.87	12.76	34.65	68	13.34	13.77	14.56	41.67
2	58	11.92	11.65	10.52	34.09	61	12.43	14.35	13.94	40.72	68	12.55	7.29	12.73	32.57	69	14.86	15.94	13.97	44.77
3	69	11.54	15.54	14.58	41.66	92	15.2	13.24	8.73	37.17	85	18.44	16.93	17.69	53.06	58	12.25	11.66	13	36.91
4	60	12.11	8.59	12.88	33.58	62	12.1	9.91	12.65	34.66	61	12.1	12.13	11.44	35.67	63	11.33	11.21	13.15	35.69
5	61	13.24	12.23	14.86	40.33	59	9.32	8.19	9.59	27.1	67	14.13	14.28	14.62	43.03	65	12.42	13.92	14.05	40.39
6	51	10.39	9.7	10.14	30.23	40	8.86	8.49	9.13	26.48	74	17.19	16.75	14.92	48.86	56	13.29	7.96	13	34.25
7	60	13.74	11.58	12.17	37.49	71	12.19	17.27	13.16	42.62	41	9.64	8.06	8.3	26	65	10.73	14.25	14.2	39.18
8	58	13.26	10.13	12.69	36.08	45	9.87	10.88	10.32	31.07	75	16.43	14.05	14.45	44.93	67	13.31	12.69	13.78	39.78
9	41	9.84	8.64	9.25	27.73	52	10.43	9.45	11.13	31.01	79	12.96	15.63	14.74	43.33	56	8.75	6.87	12.98	28.6
10	45	10.23	9.41	12.75	32.39	59	12.49	12.61	13.06	38.16	41	10.53	7.27	7.26	25.06	59	12.27	10.47	13.55	36.29
AVG	**57.5**	**12.279**	**10.869**	**11.753**	**34.901**	**59.9**	**11.444**	**11.552**	**11.279**	**34.275**	**65.3**	**13.499**	**12.326**	**12.891**	**38.716**	**62.6**	**12.255**	**11.874**	**13.624**	**37.753**

Exp. No.	Eta=6					Eta=15					Eta=10				
	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)	Time (sec)	r1 d (m)	r2 d (m)	r3 d (m)	tot d (m)
1	74	12.6	10.88	12.35	35.83	50	10	9.49	11.47	30.96	67	12.15	14.22	12	38.37
2	52	12.03	11.52	9.4	32.95	56	10.16	8.79	9.44	28.39	56	10.39	11.27	10.4	32.06
3	48	9.41	7.94	10.04	27.39	89	13.45	17.32	16.35	47.12	49	9.11	9	8.82	26.93
4	94	20.61	20.94	18.58	60.13	64	13.29	17	15.71	46	58	10.57	11.26	10.58	32.41
5	45	10.16	10.75	10.13	31.04	53	8.75	12.01	13.07	33.83	58	9.93	8.65	10.84	29.42
6	50	10.97	9.09	9.12	29.18	40	9	7.94	8.02	24.96	68	10.8	12.32	14.76	37.88
7	48	10.34	8.43	10.19	28.96	55	9.97	10.42	13.72	34.11	72	9.27	7.69	11.82	28.78
8	57	11.83	10	12.23	34.06	50	11.07	10.95	12.39	34.41	64	11.02	11.25	15.19	37.46
9	43	9.01	8.71	8.31	26.03	46	9.57	10.6	8.44	28.61	48	9.24	8.79	9.69	27.72
10	55	10.65	6.9	11.55	29.1	51	8.51	5.93	9.1	23.54	69	10.07	12.85	12.49	35.41
AVG	**56.6**	**11.761**	**10.516**	**11.19**	**33.467**	**55.4**	**10.377**	**11.045**	**11.771**	**33.193**	**60.9**	**10.255**	**10.73**	**11.659**	**32.644**

Figure 47: Raw data results for three robots real experiments

Vita

Hassan Umari was born in 1991 in Bonn, Germany. He moved with his family to Jordan, where he completed high school. In 2014 he received a B.Sc. in Mechanical Engineering from Jordan University of Science and Technology. After graduation he worked in Al-Wefaq Control Systems as an automation engineer. In 2015 he received an assistantship from the American University of Sharjah to do an M.Sc. in Mechatronics. During his studies, he also worked as a teaching and research assistant.