Efficient Frontier Detection for Robot Exploration

Matan Keidar Gal A. Kaminka

April 29, 2013

Abstract

Frontier-based exploration is the most common approach to exploration, a fundamental problem in robotics. In frontier-based exploration, robots explore by repeatedly detecting (and moving towards) *frontiers*, the segments which separate the known regions from those unknown. A *frontier detection* sub-process examines map and/or sensor readings to identify frontiers for exploration. However, most frontier detection algorithms process the entire map data. This can be a time consuming process, which affects the exploration decisions. In this work, we present several novel frontier detection algorithms that do not process the entire map data, and explore them in depth. We begin by investigating algorithms that represent two approaches: *WFD*, a graph search based algorithm which examines only known areas, and *FFD*, which examines only new laser readings data. We analytically examine the complexity of both algorithms, and discuss their correctness. We then improve by combining elements of both, to create two additional algorithms, called *WFD-INC* and *WFD-IP*. We empirically evaluate all algorithms, and show that they are all faster than a state-of-the-art frontier detector implementation (by several orders of magnitude). We additionally contrast them with each other and demonstrate the *FFD* and *WFD-IP* are faster than the others by one additional order of magnitude.

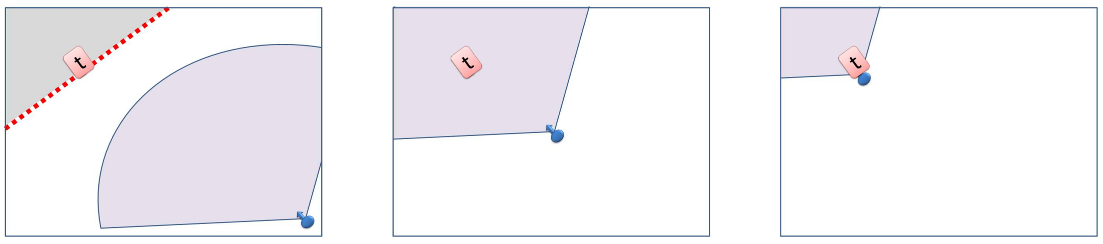
1 Introduction

The problem of exploring an unknown territory is a fundamental problem in robotics. The goal of exploration is to gain as much information as possible of the environment—through the robots' sensors—within bounded time. Applications of efficient exploration include search and rescue [19], planetary exploration [1] and military uses [16]. The most common approach to exploration is based on *frontiers*. A frontier is a segment that separates known (explored) regions from unknown regions. By moving towards frontiers, robots can focus their motion on discovery of new regions. Yamauchi [41, 42] was the first to show a frontier-based exploration strategy. His work led to many others (e.g, [6, 7, 24, 30]).

Investigations of single- and multi- robot frontier-based exploration all share the key step of *frontier detection*, which is the process that identifies the current set of frontiers, given the current map (and/or history of the robots' perceptions). In almost all papers base frontier detection on techniques borrowed from computer vision: edge detection and region extraction (see Section 2 for a detailed discussion). The frontiers need to be detected again and again, as the exploration proceeds and areas that were once unknown become known.

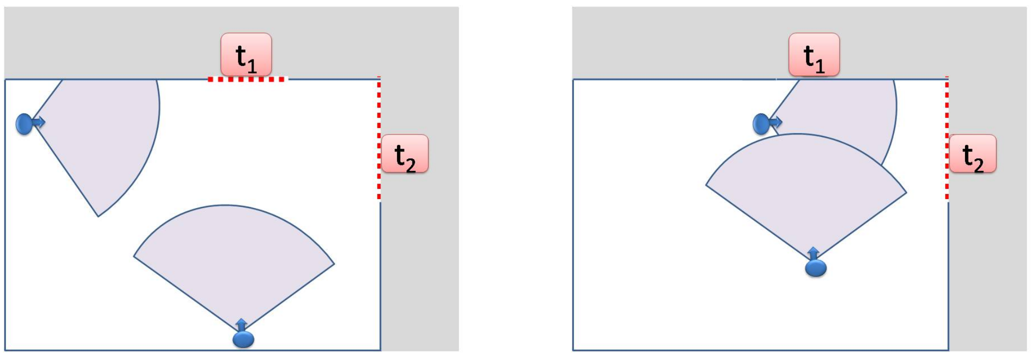
To detect frontiers, existing methods process the entire map data with every execution to the algorithm. As a result, as maps grow large, state-of-the-art frontier detection algorithms can take a number of seconds to run, even on powerful computers. This significantly impacts the exploration strategy: If a large region is explored, the robot actually has to wait in its spot until the frontier detection algorithm terminates. Therefore, most exploration implementations call the frontier detection algorithm only when the robot arrives at its destination. Others do it only once some distance has been traveled, or once some time has passed since the last detection (see 2 for details).

This slows down the exploration process, and can cause inefficiencies in the exploration. We present two examples: First, consider a common *single-robot case* (Figure 1), where a robot exploring its environment detects a frontier and moves towards it (Figure 1(a)). Because of sensor coverage, the robot may in fact sense (and clear) all remaining unknown area (Figure 1(b)), but because it cannot call the frontier-detection mechanism, it continues to move unnecessarily (Figure 1(c)). Similarly, consider a *multi-robot case* (Figure 2). Here, two robots, R_1 and R_2 which are located on bottom and top, respectively, are exploring the environment, from their initial locations (Figure 2(a)). One of the robots passes by a target assigned to the other, and thus clearing it (Figure 2(b)). But because the



(a) (b) (c)

Figure 1: A single-robot example. In 1(a) the robot is heading towards the marked target on the frontier. In 1(b) the target and all of the remaining are covered by the robot's sensors, but because the robot does not re-detect frontiers, it continues to move. In 1(c) the robot has reached the frontier, unnecessarily.



(a) (b)

Figure 2: A multi-robot example. In 2(a), the top robot (R_2) is heading towards the right target, t_2; the other robot (R_1) heads towards the top target t_1. In 2(b) R_2 has reached its target, clearing both t_1 and t_2, making R_1's movements unnecessary.

other robot cannot continuously re-detect frontiers, it unnecessarily continues towards the covered target, instead of turning to more fruitful exploration targets.

In this work, we focus on significantly speeding up the frontier detection, common to all frontier-based exploration techniques. We introduce four algorithms for fast frontier detection.

The first, *WFD* (Wavefront Frontier Detector) is an iterative method that performs a graph-search over already-visited map points (i.e., the known area), instead of the entire map (which includes also unknown areas). It builds on ideas suggested in earlier work [8] which were not evaluated as an alternative to the edge-detection state-of-the-art. However, as exploration progresses, the scanned area grows, and thus *WFD* cannot be expected to perform well in large areas.

Our second contribution is *FFD* (Fast Frontier Detector), a novel approach for frontier detection which processes raw sensor readings, and thus only scans areas that could contain frontiers. But because it works with raw sensor data, it requires extending the evolving map with additional data-structures, so that frontiers are maintained even when they are no longer within sensor range. We describe these data-structures in detail, focusing on fast implementations.

Here's another way of looking at these algorithms. *WFD* and edge-detection methods use the occupancy grid maintained by a SLAM mapper as a representation of the accumulated, processed sequence $\langle O_0, \ldots, O_t \rangle$. They thus essentially ignore most of the sequence, and act (mostly) on the latest observation of the occupancy grid G_t, as it integrates all sensor readings up to time t. The edge detection method processes G_t (the entire occupancy grid at time t). The *WFD* algorithm processes a subset, starting with the cell indicated by P_t (the latest robot position), within G_t. *FFD* uses R_t (the latest range sensor readings) directly to detect current frontiers, and a modified occupancy grid data structure to maintain/delete past frontiers. We define these terms and notations formally in Section 3.1. We then provide a theoretical complexity analysis for the algorithms, and discuss their analytical correctness.

Finally, we synthesized the different approaches of *WFD* and *FFD* into two novel algorithms: *WFD-INC* and *WFD-IP*. Both algorithms perform frontier detection on known regions but they differ from *WFD* by searching for frontiers only within the areas that were covered by the robot sensors since last call. They thus borrow from *FFD* the ability to maintain knowledge of frontiers not currently visible.

We provide a detailed evaluation of all algorithms, and contrast them with the state-of-the-art (*SOTA*). Specifically, we examine their performance in different types of environments, and on two different CPUs. We show that *WFD* and *WFD-INC* are faster than *SOTA* by 1–2 orders of magnitude, and that *FFD* and *WFD-IP* are faster than *WFD* and *WFD-INC* by additional 1–2 orders of magnitude. The results make it possible to execute real-time frontier-detection on current-day robot CPUs, opening the way to novel frontier-based exploration strategies which were impractical until now.

This paper is organized as follows. Section 2 discusses related investigations. Sections 3 and 4 discuss the *WFD* and *FFD* algorithms, respectively, including their theoretical correctness. Section 5 discusses complexity of these methods. Section 6 shows how to combine elements of *WFD* and *FFD* to create two improved algorithms, *WFD-INC* and *WFD-IP*. Section 7 reports on empirical evaluation of all algorithms, contrasting with the state of the art. Finally, Section 8 concludes.

2 Related Work

An outline of the exploration process can be described as follows: while there is an unknown territory, allocate to each robot a target point to explore and coordinate team members in order to minimize overlaps. In frontier-based exploration, targets are drawn from existing *frontiers*, segments that separate known and unknown regions (see Section 3.1 for definitions).

Most literature ignores the computational cost of frontier detection. Instead, there are two aspects that are often tackled in existing literature on exploration: deciding on next target to be explored and coordinating team members in order to minimize overlaps. The latter is not related to this work and so, we focus on the former, it may indirectly provide evidence as to the state of the art in frontier detection.

To the best of our knowledge, all of the following works utilize a standard edge-detection method for computing the frontiers. They therefore recompute target locations whenever one robot has reached its target location or whenever a certain distance has been traveled by the robots or after a timeout event.

Yamauchi [41, 42] developed the first frontier-based exploration methods. The robots explore an unknown environment and exchange information with each other when they get new sensor readings. As a result, the robots build a common map (occupancy grid) in a distributed fashion. The map is continuously updated until no new regions are found. In his work, each robot heads to the *centroid*, the center of mass of the closest *frontier*. All robots navigate to their target independently while they share a common map. Frontier detection is performed only when the robot reaches its target.

Burgard et al. [6, 7] focus their investigation on a probabilistic approach for coordinating a team of robots. Their method considers the trade-off between the costs of reaching a target and the utility of reaching that target. Whenever a target point is assigned to a specific team member, the utility of the unexplored area visible from this target position is reduced for the other team members. In their work, frontier detection is carried out only when a new target is to be allocated to a robot.

Wurm et al. [40] proposed to coordinate the team members by dividing the map into segments corresponding to environmental features. Afterwards, exploration targets are generated within those segments. The result is that in any given time, each robot explores its own segment.

In addition, Wurm [39] claims that updating frontiers frequently is important in a multi-robot setting since the map will be updated not only by the robot assigned to a given frontier, but also by all the robots in the team. In reasonably sized environments, Wurm suggests to call frontier detection on every time-step of the coordination algorithm. In real world environments, he suggests executing the frontier detection algorithm according to the traveled distance (i.e. every $0.5m - 1m$) or on every second or whenever a new target is requested.

Stachniss [32] introduced a method to make use of background knowledge about typical structures when distributing the team members over the environment. In his work, Stachniss computes new frontiers when there are new targets to be allocated. This happens whenever one robot has reached its designated target location or whenever the distance traveled by the robots or the elapsed time since last target assignment has exceeded a given threshold.

Berhault et al. [3] proposed a combinatorial auction mechanism where the robots bid on a bunch of targets to navigate. The robots are able to use different bidding strategies. Each robot has to visit all the targets that are included in his winning bid. After combining each robot's sensor readings, the auctioneer omits selected frontier cells as potential targets for the robots. Frontier detection is performed when creating and evaluating bids.

Visser et al. [37] investigated how limited communication range affects multi-robot exploration. They proposed an algorithm which takes into account wireless constraints when selecting frontier targets. Visser [36] suggests recomputing frontiers every 3–4 meters, which in his opinion, has a positive effect.

Juliá et al. [18] surveyed the most important exploration methods according to different aspects such as team coordination and integration with the SLAM algorithm. None of the compared exploration strategies takes advantage of real-time frontier detection.

Lau [24] presented a behavioral approach. The author assumes that all team members start from a known location. The team members follow the behavior and spread in the environment while updating a shared map. Frontier detection is called when the robot plans its next direction of movement.

Many other works omit details of their frontier detection timing, and thus provide no evidence for the computational cost of frontier detection. However, they broadly refer to frontier detection as a central task as part of exploration, and thus signify its centrality within modern robot exploration systems.

Sawhney et al. [30] presented an exploration method for both 2D and 3D environments. They showed a novel visibility per-time metric. Their method covers nearly the same number of points like other metric methods that can be found in literature. However, the time length of the paths is smaller. The outcome is reduced exploration time.

Simmons et al. [31] proposed to assign a target destination to each robot in a way that that maximizes the expected map knowledge over time. They proposed a bid-based heuristic. Each robot estimates its utility and cost until arriving various targets. According to this calculation, each robot creates bids. After receiving all bids, a central agent assigns a target to each robot considering minimization of the overlapping coverage of the team members.

Bouraqadi et al. [4] proposed a flocking-based approach for solving the exploration problem, where robots act according to the same set of rules. One of their rules (R5) makes the robot navigate towards the nearest frontier.

Ko et al. [20] presented a decision-theoretic approach to the mapping and exploration problem. Their approach uses an adopted version of particle filters to estimate the position in the other robot's partial map.

Fox et al. [11] presented an exploration and mapping distributed system for multi-robot exploring teams. Their system enables building a map of the environment and is robust against limited-range communication and against uncertainty of initial locations of the team members.

Zlot et al. [43] coordinates a team of exploring robots by applying a market-based approach. The market architecture maximizes the exploration information gain while minimizing the travel distance in order to maximize the overall utility.

Kuipers et al. [23] presented a method for exploration and mapping for large-scale spatial environments. Their method utilizes the qualitative properties of large-scale environments before relatively error-prone geometrical attributes. This in contrary to traditional methods which utilize sensor data to build a geometrical representation of the environment and than extract topological structure from the geometric representation.

Rekleitis et al. [28, 29] presented a strategy of localization and exploration designed for small-scale and large-scale environments. Each pair of robots that are able to sense each other work together in order to reduce localization error. They modeled the free areas as a simple polygon with holes and use a well-known planar decomposition form. For large-scale areas, they use trapezoid decomposition and for small-scale areas, they apply a triangulation of the free space.

Stachniss et al. [33] investigated the assignment of targets to the robot team members. They applied semantic information on the environment in order to better distribute the explorers over the explored environment.

Batalin et al. [2] presented an exploration method which does not utilize a map for navigating in the environment. Their algorithm uses markers which are dropped off by the robot and aid the exploration.

Puig [27] presented an algorithm for coordinating the team members during the exploration task. Their method is based on K-Means (KME) global optimization strategy.

Stoeter et al. [34] presented a mechanism for controlling a robot team for missions of exploration and surveillance. Controlling the team members is performed through a hierarchical behavior tree.

Their system is modular and enables adding new kinds of behaviors.

Ortiz et al. [22, 21] presented *Centibots*, a framework which enables to control a large team of robots performing tasks of exploration, planning and collaboration in unknown environments.

Our work on *WFD* (Section 3) is independent from previous work, though Calisi et al. [8] mentions a frontier detection algorithm that utilizes breadth-first search, similar to *WFD*. However, they do not provide details of the algorithm, nor evaluation of its performance, and so exact similarities and differences cannot be assessed.

3 Wavefront Frontier Detector

In this section we review the basic definitions and terms related to the domain of frontier-based exploration, as well as a formal definition of the frontier-based exploration problem (Section 3.1). We briefly discuss common state-of-the-art techniques for performing a Simultaneous Localization and Mapping (Section 3.2), which underly existing approaches to exploration. We then present the *WFD* algorithm which searches for frontiers on the map (Section 3.3).

3.1 Frontier-Based Exploration: Definitions

In this section we define and explain the terms that are used in the following sections. We assume the robot in question uses an occupancy-grid map representation in the exploration process (Figure 3) within the map:
Unknown Region is a territory that has not been covered yet by the robot's sensors.
Known Region is a territory that has already been covered by the robot's sensors.
Open-Space is a *known region* which does not contain an obstacle.
Occupied-Space is a *known region* which contains an obstacle.
Occupancy Grid is a grid representation of the environment. Each cell holds a probability that represents if it is occupied.
Frontier is the segment that separates known (explored) regions from unknown regions. Formally, a frontier is a set of *unknown* points that each have at least one *open-space* neighbor.

Definition. Suppose we are given a temporal sequence of observations $\langle O_0, \ldots, O_t \rangle$ (time 0 to time t), where each observation O_x is a tuple $\langle G_x, P_x, R_x \rangle$ composed of: (i) the occupancy-grid G_x of time x; (ii) the robot pose P_x (in occupancy-grid coordinates); and (iii) the range sensor readings R_x originating at the robot location (given in either ego-centric polar coordinates, or in occupancy-grid coordinates). The *Frontier Detection Problem* is to return all frontiers existing at time t, given the sequence.

Existing algorithms for frontier detection rely on edge-detection methods. The algorithms systematically search for frontiers all over the occupancy-grid, i.e., both in known and unknown regions.

3.2 Simultaneous Localization and Mapping Methods

Simultaneous Localization and Mapping (*SLAM*) is a method utilized by autonomous mobile robots for building a map of an unknown environment while at the same time navigating the environment using the map. The term *SLAM* was originally developed by Leonard et al. [25, 10]. In *SLAM* both the trajectory of the mobile robot, its location and the location of the landmarks observed by the robot are estimated without using any *a priori* knowledge of the environment. *SLAM* is a concept and hence, there are various algorithms in literature which deal with the *SLAM* problem:

Extended Kalman Filter The *Kalman Filter* (*KF*) is a method for implementing Bayes filters. It was invented by Rudolph Emil Kalman [38, 35] to apply filtering and perform prediction in linear systems. In Kalman Filter, beliefs are represented by moments (i.e., by the mean and the covariance). Markov assumptions are held and the posteriors are Gaussian. Kalman filters are based on *linear* system dynamics. However, real-world systems are seldom linear in practice (e.g., robot which moves in a cyclic path). The *Extended Kalman Filter* (*EKF*) uses a non-linear functions that describe the system dynamics. In *EKF-SLAM* systems, there is only one map (part of the system state) that is composed from the observed landmarks.

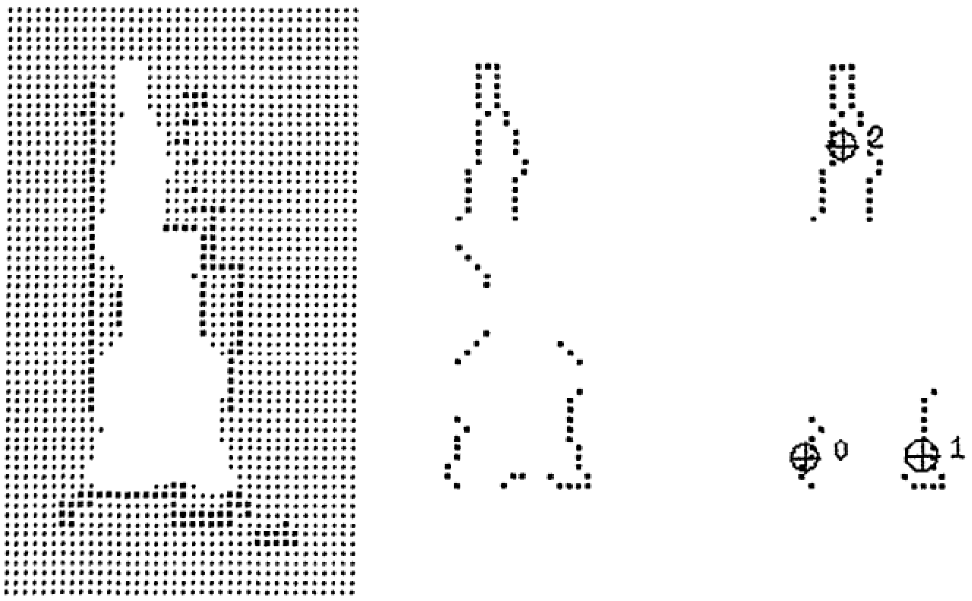


Figure 3: Evidence grid, frontier points, extraction of different frontiers (from left to right). Taken from [42].

Particle Filter A *Particle Filter* (also known as *sequential Monte Carlo method*) is a model estimation method which is based on simulation. Particle Filter is the online version of *Markov chain Monte Carlo* (*MCMC*) methods [15, 12, 26]. With sufficient samples, a particle filter is often an alternative to *EKF* with the advantage that it approaches the Bayesian optimal estimate and so it is more accurate than *EKF*. Each particle is a possible hypothesis of what is the true world state at a certain time, that is, each particle is an instance of the system state at a certain time. By state we refer to a collection of variables such as the map of the explored environment, the robot position etc.

In this work we focus on particle filter systems since in practice, they provide more accurate results.

3.3 *WFD*

We present a graph search based approach for frontier detection. The algorithm, *WFD* (Algorithm 3.1–3.2), processes the points on map which have already been scanned by the robot sensors and therefore, does not always process the entire map data in each run, but only the known regions.

WFD (Algorithm 3.1) is based on Breadth-First Search (*BFS*). First, the occupancy-grid point that represents the current robot position is enqueued into $queue_m$, a queue data-structure used to determine the search order (Algorithm 3.1 lines 1–3).

Next, a *BFS* is performed (Algorithm 3.1 lines 4–22) in order to find all frontier points contained in the map. The algorithm keeps scanning only points that have not been scanned yet and represent open-space (Algorithm 3.1 line 16). The above scanning policy ensures that only known regions (that have already been covered by the robot's sensors) are actually scanned. The significance of this is that the algorithm does not have to scan the entire occupancy-grid each time.

Because frontier points are adjacent to open space points, all relevant frontier points will be found when the algorithm finishes (Algorithm 3.1 line 22). If a frontier point is found, a new *BFS* is performed in order to extract its frontier (Algorithm 3.2 lines 6–23). This *BFS* searches for frontier points only. Extracting the frontier is ensured because of the connectivity of frontier points.

At the end of the frontier extraction process (Algorithm 3.2 line 23), the extracted frontier data is saved to a set data-structure that stores all frontiers found in the algorithm's run.

In order to avoid rescanning the same map point and detecting the same frontier reachable from two frontier points, *WFD* marks map points with four indications:

1. Map-Open-List: points that have already been enqueued by the outermost *BFS* (Algorithm 3.1 line 17)

Algorithm 3.1 Wavefront Frontier Detector (WFD)

```
Require: $queue_m$ // queue, used for detecting frontier points from a given map
Require: $pose$ // current global position of the robot
 1: $queue_m \leftarrow \emptyset$
 2: ENQUEUE($queue_m$, $pose$)
 3: mark $pose$ as "Map-Open-List"

 4: while $queue_m$ is not empty do
 5:     $p \leftarrow$ DEQUEUE($queue_m$)
 6:     if $p$ is marked as "Map-Close-List" then
 7:         continue
 8:     end if
 9:     if $p$ is a frontier point then
10:         $NewFrontier \leftarrow$ EXTRACT-FRONTIER-2D($p$)
11:         save $NewFrontier$
12:         mark all points of $NewFrontier$ as "Map-Close-List"
13:     end if

14:     for all $v \in N(p)$ do // get all neighbors of current point
15:         $isNeighbor \leftarrow v$ has at least one "Map-Open-Space" neighbor
16:         if $v$ not marked as one of {"Map-Open-List","Map-Close-List"} and $isNeighbor$ then
17:             ENQUEUE($queue_m$,$v$)
18:             mark $v$ as "Map-Open-List"
19:         end if
20:     end for

21:     mark $p$ as "Map-Close-List"
22: end while
```

Algorithm 3.2 Extract Frontier 2D

```
Require: $p$ // frontier point
Ensure: $NewFrontier$ // extracted frontier which was found starting from $p$
 1: function EXTRACT-FRONTIER-2D($p$)
 2:     $queue_f \leftarrow \emptyset$
 3:     $NewFrontier \leftarrow \emptyset$
 4:     ENQUEUE($queue_f, p$)
 5:     mark $p$ as “Frontier-Open-List”

 6:     while $queue_f$ if not empty do
 7:         $q \leftarrow$ DEQUEUE($queue_f$)
 8:         if  $q$ is marked as one of {“Map-Close-List”,”Frontier-Close-List”} then
 9:             continue
10:         end if
11:         if $q$ is a frontier point then
12:             add $q$ to $NewFrontier$
13:             for all $w \in N(q)$ do // get all neighbors of current frontier point
14:                 $tags \leftarrow$ {“Frontier-Open-List”,“Frontier-Close-List”, “Map-Close-List”}
15:                 if $w$ not marked as one of $tags$ then
16:                     ENQUEUE($queue_f, w$)
17:                     mark $w$ as “Frontier-Open-List”
18:                 end if
19:             end for
20:         end if
21:         mark $q$ as “Frontier-Close-List”
22:     end while
23:     return $NewFrontier$
24: end function
```

2. Map-Close-List: points that have already been dequeued by the outermost *BFS* (Algorithm 3.1 line 5)
3. Frontier-Open-List: points that have already been enqueued by the frontier extraction *BFS* (Algorithm 3.2 line 16)
4. Frontier-Close-List: points that have already been dequeued by the frontier extraction *BFS* (Algorithm 3.2 line 7)

The above marks indicate the status of each map point and determine if there is a need to handle it in a given time.

The key innovation in *WFD* is that it prevents scanning unknown regions, since frontiers never appear there. However, it still searches all known space.

4 Fast Frontier Detector

We present a novel approach to the problem of frontier detection. The *FFD* algorithm (Section 4.1) avoids searching for frontiers both in known and unknown regions of the map; it only searches in the boundary between them. This significantly reduces the search area. However, the search-space reduction forces *FFD* to run in the background persistently. In order to be robust against map orientation changes caused by loop closures, *FFD* has to perform maintenance over previously detected frontiers (Section 4.1.4). At the end of this section, we prove that *FFD* is complete and sound (Section 4.2).

4.1 *FFD*

Unlike other frontier detection methods (including *WFD*), our proposed algorithm (Algorithm 4.1) only processes new laser readings which are received in real time. It therefore avoids searching both known and unknown regions. In doing this, we make use of the fact that by definition, frontiers represent the boundaries between the known and unknown regions of the environment (see Figure 3). Hence, scanning all unknown regions is definitely unnecessary. The *FFD* algorithm contains four steps (Algorithm 4.1), and can be called with every new scan.

Algorithm 4.1 Fast Frontier Detector (FFD): Outline

```
Require: $frontiersDB$ // data-structure that contains last known frontiers
Require: $pose$ // current global position of the robot
Require: $lr$ // laser readings which were received in current iteration. Each element is a 2-d cartesian
    point
Require: $activeArea$ // data-structure that contains all points that lie inside the active area

1: // polar sort readings according to robot position
2: $sorted \leftarrow$ SORT-POLAR($lr, pose$)

3: // get the contour from laser readings
4: $contour \leftarrow$CONTOUR($sorted$)

5: // extract new frontiers from contour
6: $NewFrontiers \leftarrow$ EXTRACT-FRONTIERS-1D($contour$)

7: // maintainance of previously detected frontiers
8: MAINTAIN-FRONTIERS($NewFrontiers, frontiersDB, activeArea$)
```

4.1.1 Sorting

The first step (Algorithm 4.2) sorts range readings based on their angle, i.e., based on the ego-centric polar coordinates with the robot as the origin. Normally, laser readings are given as a sorted set of polar coordinated points, making this sorting step unnecessary. However, if this is not the case, a sorting is needed to be applied on the received laser readings.

In this case, we assume that range readings are a set of Cartesian coordinated points, which consists of the locations of range hits ($\{(x_0, y_0), \ldots, (x_n, y_n)\}$ where n is the number of readings). The naive method for converting Cartesian to polar coordinates uses two CPU time-consuming functions: *atan2* and *sqrt*.

To speed angle sorting, we use a cross product [9] to avoid converting Cartesian to polar coordinates, while still sorting the points based on polar angle. Given 3 Cartesian coordinated points:

$$P_0 = (x_0, y_0), P_1 = (x_1, y_1), P_2 = (x_2, y_2)$$

the *cross product* is defined as:

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0) \cdot (y_2 - y_0) - (x_2 - x_0) \cdot (y_1 - y_0)$$

If the result is positive, then $\overrightarrow{P_0P_1}$ is clockwise from $\overrightarrow{P_0P_2}$. Else, it is counter-clockwise. If the result is 0, then the two vectors lie on the same line in the plane (i.e., the angle is the same).

Therefore, by examining the sign of the cross product, we can determine the order of the Cartesian points according to polar coordinates, without calculating their actual polar coordinate value. This applies only five subtractions and two multiplications which are far less time-consuming than calling *atan2* and *sqrt*.

Algorithm 4.2 Fast Frontier Detector (FFD): Sorting Stage

```
Require: $lr$ // set of laser readings
Require: $pose$ // current robot position
Ensure: $sorted$ // sorted laser readings according to polar coordinates
 1: function POLAR-SORT($lr, pose$)
 2:     $sorted \leftarrow \emptyset$
 3:     $A_r \leftarrow \emptyset$
         // set robot position as origin
 4:     for all Point $p \in lr$ do
 5:         $A_r \leftarrow polar \cup \{(p.x - pose, x, p.y - pose.y)\}$
 6:     end for
 7:     $sorted \leftarrow$ SORT($A_r$, POLARCOMPARATOR)
 8:     return $sorted$
 9: end function

10: function POLARCOMPARATOR($p_1, p_2$)
11:     $s \leftarrow (p_1.x \cdot p_2.y - p_2.x \cdot p_1.y)$ // calculate cross-product of input points
12:     if $s > 0$ then // check the sign of the cross-proudct
13:         return 1
14:     else if $s < 0$ then
15:         return -1
16:     else
17:         return 0
18:     end if
19: end function
```

4.1.2 Contour

In this step (Algorithm 4.3 lines 2–10) we use the angle-sorted laser readings. The output of the contour step is a contour which is built from the sorted laser readings. The algorithm computes the line that lies between each two adjacent points from the set. The line is computed by calling the function GET-LINE. In our implementation we use *Bresenham's line algorithm* [5]. Next, all points that are represented by all the lines (including the points from the laser readings set) are merged into a contour (Figure 4).

Algorithm 4.3 Fast Frontier Detector (FFD): Contour Stage

```
Require: $sorted$ // sorted set of laser readings
Ensure: $contour$ // the contour built from the sorted laser readings
 1: function CONTOUR($sorted$)
 2:     $prev \leftarrow$ POP($sorted$)
 3:     $contour \leftarrow \emptyset$
 4:     for all Point $curr \in sorted$ do
 5:         $line \leftarrow$ GET-LINE($prev, curr$)
 6:         for all Point $p \in line$ do
 7:             $contour \leftarrow contour \cup \{p\}$
 8:         end for
 9:     end for
10:     return $contour$
11: end function
```

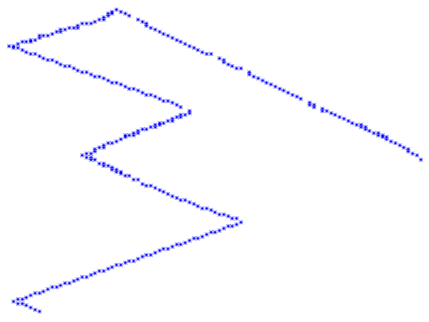


Figure 4: Example of produced contour.

4.1.3 Detecting New Frontiers

In this step (Algorithm 4.4 lines 2–21) the algorithm extracts new frontiers from the previously calculated contour. There are three cases correspond to each two adjacent points in the contour:

1. **Current scanned point is not a frontier cell.** Therefore, it does not contribute any new information about frontiers and can be ignored (lines 8–11).
2. **Current and previous scanned points are frontier cells.** Therefore, both points belong to the same frontier and current scanned point is added to last detected frontier (lines 12–14).
3. **Current point is a frontier cell but the previous is not.** A new starting point of a frontier was detected. Hence, the algorithm creates a new frontier and adds the new starting point to it (lines 15–18).

Algorithm 4.4 Fast Frontier Detector (FFD): Extract Frontier Stage

```
Require: $contour$ // the contour built from the sorted laser readings
Ensure: $NewFrontiers$ // list of new detected frontiers that lie on the contour
 1: function EXTRACT-FRONTIERS-1D($contour$)
 2:     $NewFrontiers \leftarrow \emptyset$ // list of new extracted frontiers
 3:     $prev \leftarrow POP(contour)$
 4:     if $prev$ is a frontier cell then // special case
 5:         create a new frontier in $NewFrontiers$
 6:     end if
 7:     for all Point $curr \in contour$ do
 8:         if $curr$ is not a frontier cell then
 9:             $prev \leftarrow curr$
10:         else if $curr$ has been visited before then
11:             $prev \leftarrow curr$
12:         else if $curr$ and $prev$ are frontier cells then
13:             add $curr$ to last created frontier
14:             $prev \leftarrow curr$
15:         else
16:             create a new frontier in $NewFrontiers$
17:             add $curr$ to last created frontier
18:             $prev \leftarrow curr$
19:         end if
20:     end for
21:     return $NewFrontiers$
22: end function
```

Avoiding Errors in Frontier Extraction Algorithm 4.4 lines 10–11 require some special attention. *FFD*'s frontier extraction (Algorithm 4.4) works by traversing the contour generated in the previous step. This contour has been generated based on the laser readings, using synthetic lines to connect points which correspond to laser readings at different angels. In other words, possible "holes" in the contour have been masked by synthetic lines.

Because of this, under some specific conditions, points that lie on the synthetic lines can meet the frontier criteria, erroneously. If *curr* is a point on the synthetic line, then its occupancy value will be marked as *unknown*, regardless of its real-world status. And as a result, if it has an open-space neighbor, it will be marked as a frontier. Because the frontier extraction process works directly with sensor readings, not with the map, this will happen even if in previous scans through the area (say, from a different position), the same point has been declared *known* in the map.

To address this potential error, we keep track of sensor visits (sensor covers) of each map cell. The definition of a frontier point is now expanded: a frontier point is a point whose occupancy value is unknown (this occupancy value is given by the sensor), has at least one open-space neighbor, and has not been visited before (this information is what we keep track of in the map). Given a contour, the detection of new frontiers ignores points that have already been scanned by the laser sensors and treats them as non-frontier points.

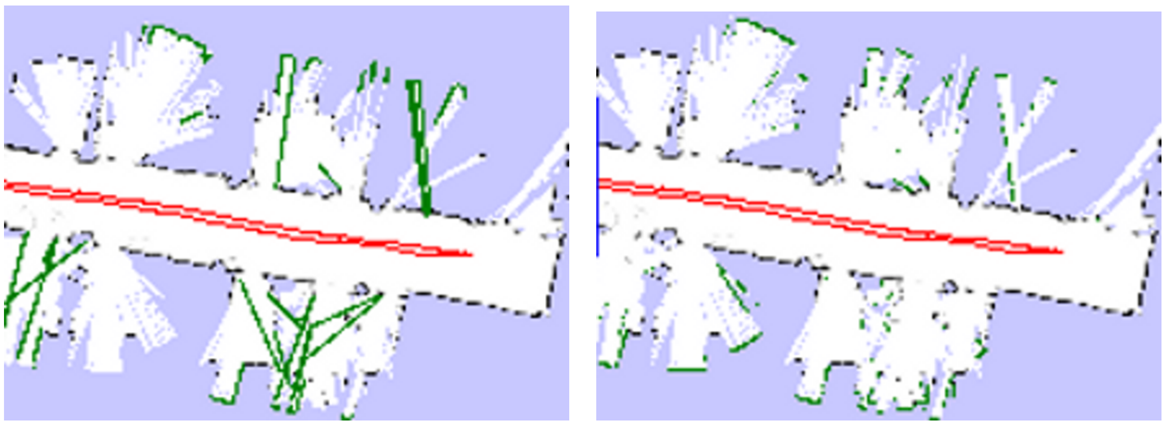

(a) Incorrectly re-detected frontiers. (b) Correct detection.

Figure 5: An example of re-detecting same frontiers.

Figure 5 demonstrates the necessity of the above. Figure 5(a) shows frontiers extraction without tracking the number of visits, from early runs of *FFD*, before this subtle error was revealed and corrected. It can be seen that there are frontiers that lie inside an open space area. This is absolutely wrong because frontiers are supposed to be positioned on the boundaries between known and unknown regions. In contrast, Figure 5(b) shows frontiers extraction with the problem corrected. It can be seen that every frontier separates known and unknown regions.

4.1.4 Maintaining Previously Detected Frontiers

FFD gains its speed by processing the laser readings only, rather than entire regions of the map. However, if the robot navigates towards a specific frontier, other previously detected frontiers are no longer updated because they are not covered by the robot's sensors. Thus, scanning the new received laser readings enables *FFD* to detect only *new* frontiers in each execution. In this step (Algorithm 4.5 lines 2–20), in order to get complete information about the frontiers, the algorithm performs maintenance over previously detected frontiers which are no longer covered in the range of the sensors. Only by joining together new detected frontiers and previously detected frontiers, we get the overall frontiers in current world state. This step has multiple goals: avoiding detection of new frontiers in an already scanned area (Section 4.1.3), eliminating frontier points which are no longer belong to frontiers (Section 4.1.4) and joining correctly the new detected frontiers together with previously detected frontiers (Section 4.1.4).

The *FFD* Data-Structures In order to perform the maintenance step within a very short time as possible, *FFD* utilizes two data-structures which have a short access time. These data structures must maintain memory of frontiers between calls. Thus *FFD* has to have persistent memory, i.e., data structures that persist between calls. This is contrast to other approaches that can be executed in a certain time, and only then.

Another thing to note is that in particle filter based systems (our focus in this work), each particle represents a possible hypothesis of the world state (including the robot position of course). The "best" particle is chosen according to a likelihood measurement. *FFD* requires the previously detected frontiers to be robust against map orientation changes caused by loop-closures of the mapping algorithm. Therefore, when a new laser reading is received, *each* particle executes its own instance of *FFD* algorithm on its own map, using its own data structures. More specifically, each particle performs maintenance with its own map because particles do not share maps. We describe the data structures for maintenance below.

The first, *Grid of Frontier Indices*, is an extension of the occupancy grid (though it can be implemented as a separate entity). In addition to occupancy information, each grid cell contains *a frontier index*, pointing to a frontier to which the grid cell belongs, or NULL otherwise. The pointer points to a record stored in the Frontier Database (described below). In our implementation, we used integer index values. After accessing a grid cell (Algorithm 4.5 Line 3), querying for its frontier index is $O(1)$.

The second, *Frontier Database*, maps frontier indices (pointers) to sets of points ($frontierDB$

Algorithm 4.5 Fast Frontier Detector (FFD): Maintenance of Previously Detected Frontiers

```
Require: $NewFrontiers$ // list of new detected frontiers
Require: $frontiersDB$ // data-structure that contains last known frontiers
Require: $activeArea$ // data-structure that contains all points that lie inside the active area
 1: function MAINTAIN-FRONTIERS($NewFrontiers, frontiersDB, activeArea$)
 2:     for all Point $p \in ActiveArea$ do // eliminate previously detected frontiers
 3:         if $p$ is a frontier cell then
 4:              // split the current frontier into two partial frontiers
 5:             get the frontier $f \in frontiersDB$ which enables $p \in f$
 6:             $f_1 \leftarrow f[1 \dots p]$
 7:             $f_2 \leftarrow f[(p+1) \dots |f|]$
 8:             remove $f$ from $frontiersDB$
 9:             add $f1$ and $f2$ to $frontiersDB$
10:         end if
11:     end for
12:     for all Frontier $f \in NewFrontiers$ do // store new detected frontiers
13:         if $f$ overlaps with an existing frontier $ExistFrontier$ then
14:             $merged \leftarrow f \cup ExistFrontier$
15:             remove $ExistFrontier$ from $frontiersDB$
16:             add $merged$ to $frontiersDB$
17:         else
18:             create a new index and add $f$ to $frontiersDB$
19:         end if
20:     end for
21: end function
```

requirement of Algorithm 4.1 and Algorithm 4.5). All detected frontiers are stored in this data-structure. We use it to map frontier index to the actual set that contains the points in world coordinates. In our implementation, we use the default C++ implementation of a map template, which is implemented as a self-balancing binary search tree. Therefore, assuming n represents the number of frontiers stored in the database, searching for a frontier index takes $O(\log n)$, inserting a new frontier takes $O(\log n)$ and removing a frontier index takes $O(\log n)$, though a (hash) table lookup implementation can make this faster.

Eliminating Previously Detected Frontiers In order to complete the process, points which are no longer in frontiers (i.e. were covered by the robot's sensors) have to be eliminated. Lines 2–11 in Algorithm 4.5 contain the elimination logic applied by *FFD*.

Let t_i be a time frame and lr_{t_i} be the laser readings which were received in time frame t_i. In order to perform maintenance in a specific time, we define the *Active Area* of time frame t_i to be the blocking rectangle that can be constructed using the farthest laser readings of lr_{t_i}, relative to the robot position in time frame t_i.

$$x_{min} = min(\{x|x \in lr_{t_i}\}), y_{min} = min(\{y|y \in lr_{t_i}\})$$

$$x_{max} = max(\{x|x \in lr_{t_i}\}), y_{max} = max(\{y|y \in lr_{t_i}\})$$

$$ActiveArea_{t_i} = \{(x,y)\,|x_{min} \leq x \leq x_{max}, y_{min} \leq y \leq y_{max}\}$$

The active area's rectangle is constructed from the following vertices: (x_{min}, y_{min}), (x_{min}, y_{max}), (x_{max}, y_{max}), (x_{max}, y_{min}). The rectangle is an approximation to the real active area that is actually bounded within the laser readings.

By processing received laser readings, *FFD* extracts new frontiers. However, in order to get the complete world's frontiers state, points that are no longer on frontiers have to be eliminated. *FFD* maintains a frontier database which maps an integer (frontier index) to a set of points (frontier).

An unknown region is classified as known region only if it is covered by the robot's sensors. *FFD* gets its input from the new received laser readings, and thus only regions that are covered by the robot's sensors might contain frontiers that have to be eliminated. Thus, if there are frontiers that need to be eliminated, they must lie inside the *Active Area*. Hence, the active area is a key feature in the process of maintaining frontiers. *FFD* scans each point that lies inside the active area and checks if it was previously belonged to a frontier. The check can be performed very fast as explained before. If the current scanned point belonged to a frontier, the current scanned point is removed from the frontier and the frontier is split into two partial frontiers using the current scanned point as a pivot (Algorithm 4.5 lines 6–7).

In the end of this process, all no-longer frontier points in the frontier database are removed and the database contains only points that are still valid frontiers.

Storing New Detected Frontiers In the last phase of the maintenance step (Algorithm 4.5 lines 12–20) new detected frontiers are stored in the frontier database alongside with existing valid frontiers. For each new detected frontier, *FFD* checks if it overlaps with an already existing frontier. This comparison can be performed in a short time using the matrix of frontier indices. Each frontier point is queried in $O(1)$ operations. If an overlap is found, the frontier is merged with the frontier that it is overlapped with. If no overlap is found, then the frontier is inserted to the frontier database.

4.2 *FFD* is Sound and Complete

We show that Algorithm 4.1 is sound and complete. We begin with a lemma that demonstrates that *FFD* always recognizes new frontiers (i.e., frontiers that appeared at time t, but did not exist before). This will then be used to prove completeness of *FFD*.

Lemma 4.1. *Suppose f is a frontier point at time t, which was not a frontier point at any time s, where $s < t$. Then FFD will mark f as a frontier given observation O_t.*

Proof. Let f be a valid frontier point in time t and was not classified as frontier in time $s < t$. Since f is a valid frontier point, then it has a value of *Unknown* and has at least one *Open Space* neighbor at time t. Assume towards a contradiction that *FFD* did not recognize f as a frontier point. First, let us show that f is contained in the contour handled in Algorithm 4.4 lines 2–21. Since f is a valid frontier point, then it has a value of *Unknown* and has at least one *Open Space* neighbor in time t. The point f cannot be located wholly within an *unknown* region because it must have at least one *Open Space* neighbor. Also, the point f cannot be located wholly within a *known* region since f is a valid frontier point and hence, its value is *Unknown*. Therefore, f must be located on the contour itself. Lines 2–21 in Algorithm 4.4 handle points on the contour, which we have just shown f is on. In these lines, the *FFD* algorithm scans *all* contour points sequentially and specifically searches for frontier points. Because it scans *all* points on the contour, and we have shown that f is on the contour, it follows that f would be detected, contradicting the assumption that *FFD* did not recognize f as a frontier point at time t. □

We now turn to proving the completeness of the *FFD* algorithm.

Theorem 4.2. *Let f be a valid frontier point at time t. Then FFD will mark f as a frontier point given the sequence of observations $\langle O_0, \ldots, O_t \rangle$.*

Proof. Two cases should be examined:
Case 1. f is a new frontier point at time t. Trivially, this case is handled directly by lemma 4.1.
Case 2. f was a new frontier point at time s, where $s < t$. Let s be the earliest time in which f was a frontier. Based on lemma 4.1, it follows that it was detected at this time. All that remains to show is that given f is still valid at time t, *FFD* will maintain knowledge of it from time s and report on it. If f is still a valid frontier point at time t, then it has not been covered yet by the robot's sensors. Otherwise, it would no longer contain an *Unknown* value and hence, would not be a valid frontier point. So if it was not yet covered, it must be a frontier point that is maintained by *FFD*. The only way in which f can be eliminated from being classified as a frontier point is done by Algorithm 4.5 lines 2–11. In these lines, *FFD* scans all points that are covered by the robot's sensors and checks if any points should be eliminated (Algorithm 4.5 line 3). Since f is not covered by the sensors, then it will not be scanned and eliminated in time $t \Rightarrow f$ remains classified as a frontier by *FFD*.

In both cases we show *FFD* will recognize f to be a valid frontier at time t. Since Theorem 4.2 is true for any frontier point valid at time t, it follows that *FFD* is complete. □

To show the soundness of *FFD*, we must demonstrate that there does not exist a case where *FFD* marks a point $\hat{f}$ as a frontier, when it is not.

Theorem 4.3. *Let $\hat{f}$ be an arbitrary point in the occupancy grid, which is* not *a frontier at time t. Then FFD will not return $\hat{f}$ as a frontier point, given the sequence of observations $\langle O_0, \ldots, O_t \rangle$.*

Proof. Assuming that $\hat{f}$ is an arbitrary point which is *not* a frontier point at time t, then $\hat{f}$ is either contains value different from *Unknown* or all its adjacent values are different from *Open Space*. We will examine two cases:
Case 1. $\hat{f}$ is marked as a new frontier. Suppose, towards a contradiction, that *FFD* detects $\hat{f}$ as a new frontier (i.e., true at time t, but not a frontier in time s, where $s < t$). Since detection of new frontier points (Algorithm 4.4 lines 2–21) considers only points on the contour, it follows that $\hat{f}$ must be located on the contour *and* detected by Algorithm 4.4 lines 2–21. However, Algorithm 4.4 line 8 specifically avoids classifying non-frontier points as frontiers. Since $\hat{f}$ is a non-frontier point, it is ignored by *FFD*. Therefore, $\hat{f}$ cannot be marked as a new frontier $\Rightarrow$ contradicting the assumption that it is detected by *FFD* as a new frontier. Case 1 is impossible.
Case 2. $\hat{f}$ is an old frontier but was not eliminated by the maintenance routine. Suppose, towards a contradiction, that $\hat{f}$ is located inside the active area and is not eliminated by the maintenance section. Therefore, $\hat{f}$ is a point that was covered by the robot's sensors and no longer contains an *Unknown* value, yet is still marked as a frontier by the *FFD* algorithm. We remind the the reader that in order to maintain frontier points across runs, each point in the grid keeps a value which contains NULL if the point is not a frontier point or the index of the frontier to whom it belongs. Therefore, in Algorithm 4.5 line 3 *FFD* scans all points in the active area and checks if they contain a frontier index. When *FFD* scans $\hat{f}$, it finds out that it contains a valid frontier index (because it has previously belonged to a valid frontier) and continues executing Algorithm 4.5 lines 5–9. In these lines, *FFD* checks and removes from the frontier database all points that are no longer frontier points and previously were frontier points. Thus, $\hat{f}$ will be eliminated after scanning the active area, contradicting the assumption that $\hat{f}$ was not eliminated.

Since in both cases we show that *FFD* necessarily eliminated $\hat{f}$ from the valid frontier list, it follows that if $\hat{f}$ is not a frontier-point at time t, it would not be marked as such by *FFD*. Since Theorem 4.3 holds for any arbitrary point, it follows that *FFD* never incorrectly marks a non-frontier point as a frontier. It is thus sound. □

5 Complexity Analysis

In this section, we show a theoretical complexity analysis of the *WFD* and *FFD* algorithms. In Section 5.1, we discuss *WFD* and in Section 5.2, we discuss *FFD*.

5.1 *WFD* Complexity Analysis

As shown in Section 3, *WFD* is based on Breadth-First Search (*BFS*) over the occupancy grid. Within every call to *WFD*, it scans all *open-space* regions for frontier points. When a frontier point is found, *WFD* performs another Breadth-First Search in order to extract its frontier. Regarding a graph $G = (V, E)$, the upper bound of *BFS* run-time is $\mathcal{O}(E + V)$, full proof can be found in [9]. Therefore, complexity of scanning the *open-space* regions is linear in size of the area of the *open-space* regions. Moreover, frontier points are always located on the edges of the *open-space* regions. Thus, extracting frontiers from given frontier points is also linear in size of the perimeter of the *open-space* region. Hence, *WFD*'s run-time complexity is linear in size of the area (denoted $\mathcal{S}(\cdot)$) and perimeter ($\mathcal{P}(\cdot)$) of the *open-space* region and can be formulated as:

$$\mathcal{O}\left(\underbrace{\mathcal{S}(open - space)}_{area} + \underbrace{\mathcal{P}(open - space)}_{perimeter}\right) \tag{1}$$

In the following sections we discuss *WFD*'s best case and worst case. The reader should be aware to the fact that for a given map and a time stamp, the *open-space* area is determined by the trajectory of the robot. However, its perimeter length affects the run-time and two cases should be examined:

5.1.1 *WFD* Best Case

The perimeter of the *open-space* regions is minimal relatively to the area of the *open-space* regions. This can be shown in Figure 6. In this case we denote the area of the shape as S_{open} and its perimeter as P_{opt}. Their values are shown in Equations (2) and (3):

$$S_{open} = 4\pi r^2 \tag{2}$$

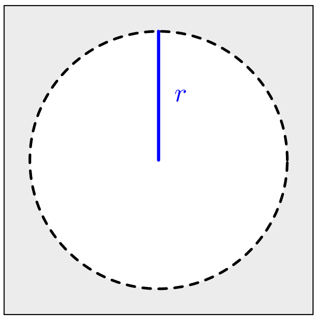


Figure 6: *WFD* Best Case: perimeter of *open-space* regions is small as possible

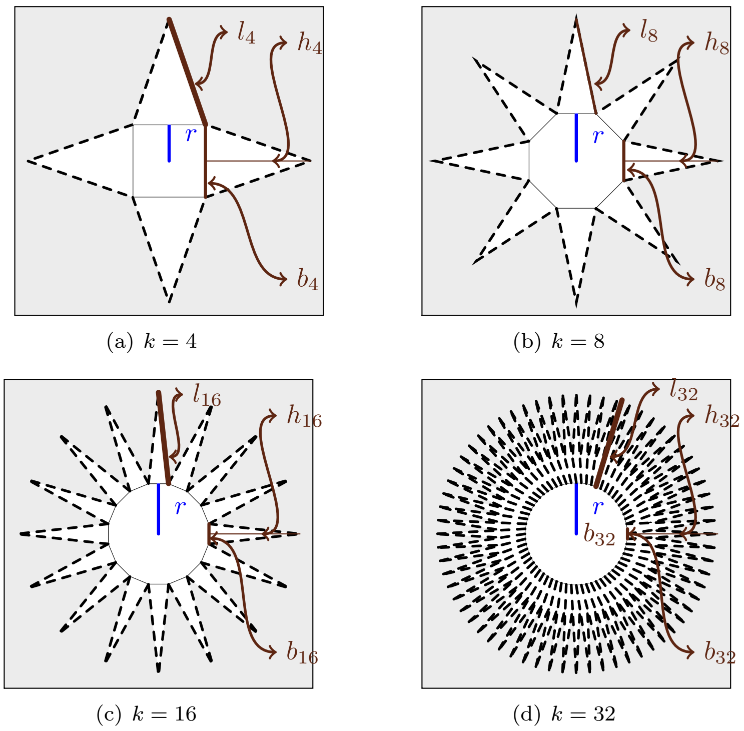


Figure 7: *WFD* worst case: different values of k

$$P_{opt} = 2\pi r \tag{3}$$

5.1.2 *WFD* Worst Case

In the worst case we would like to maximize the length of the perimeter while keeping the total area of the *open-space* regions. Therefore we use a polygon as an approximation to a circle. We take an inner regular polygon and build triangles on each side. The level of accuracy is determined by k, the number of vertices. For a given k value we denote the total area of the shape by S_k and the total perimeter length by P_k. In order to bound an area, at least 3 vertices have to be used. As previously mentioned, all areas are equal and thus, for each $k \in \mathbb{N} \setminus \{0, 1, 2\}$ holds: $S_k = S_{open}$. Figure 7 demonstrates the results of different k values.

5.1.3 The General Case

Assuming we have a general shape with an arbitrary number of vertices, k and a radius, r. Let b_k be the base of the inner polygon that is located inside the *open-space* regions (examples can be found in Figure 7). The length of base b_k is given by the formula:

$$b_k = 2r \cdot \tan\frac{\pi}{k} \tag{4}$$

Let h_k be the height of an outer triangle. The length of h_k is given by equation (5):

$$\begin{aligned} S_k &= S_{open} \\ k \cdot \frac{r \cdot b_k}{2} + k \cdot \frac{h_k \cdot b_k}{2} &= 4\pi r^2 \\ k \cdot r \cdot b_k + k \cdot h_k \cdot b_k &= 8\pi r^2 \\ h_k = \frac{8\pi r^2 - k \cdot r \cdot b_k}{k \cdot b_k} &= \frac{8\pi r^2}{k \cdot b_k} - r \end{aligned} \tag{5}$$

Let l_k be the length of an outer triangle side. The length of l_k is given by equation (6):

$$l_k = \sqrt{(h_k)^2 + \left(\frac{b_k}{2}\right)^2} \tag{6}$$

Therefore, the length of the polygon's perimeter equals to the total length of the outer edges of the triangles. We can calculate the perimeter of the polygon by:

$$P_k = 2k \cdot l_k \tag{7}$$

We now express the perimeter P_k as a function of the area S_{open}:

$$\begin{aligned} P_k &= 2k \cdot l_k \\ &= 2k\sqrt{(h_k)^2 + \left(\frac{b_k}{2}\right)^2} \\ &= 2k\sqrt{\left(\frac{8\pi r^2}{k \cdot b_k} - r\right)^2 + \left(\frac{2r \cdot \tan\frac{\pi}{k}}{2}\right)^2} \\ &= 2k\sqrt{\left(\frac{8\pi r^2}{k \cdot b_k} - r\right)^2 + \left(r \cdot \tan\frac{\pi}{k}\right)^2} \end{aligned} \tag{8}$$

Equations (2) and (4) can be joint with Equation (8):

$$\begin{aligned} P_k &= 2k\sqrt{\left(\frac{2S_{open}}{k \cdot 2r \cdot \tan\frac{\pi}{k}} - r\right)^2 + \left(r \cdot \tan\frac{\pi}{k}\right)^2} \\ &= 2k\sqrt{\left(\frac{S_{open}}{k \cdot r \cdot \tan\frac{\pi}{k}} - r\right)^2 + \left(r \cdot \tan\frac{\pi}{k}\right)^2} \\ &= 2k\sqrt{\left(\frac{S_{open}}{k \cdot r \cdot \tan\frac{\pi}{k}}\right)^2 - 2\frac{r \cdot S_{open}}{k \cdot r \cdot \tan\frac{\pi}{k}} + r^2 + r^2 \cdot \tan^2\frac{\pi}{k}} \\ &= 2k\sqrt{\left(\frac{S_{open}}{k \cdot r \cdot \tan\frac{\pi}{k}}\right)^2 - 2\frac{S_{open}}{k \cdot \tan\frac{\pi}{k}} + r^2\left(1 + \tan^2\frac{\pi}{k}\right)} \\ &= 2k\sqrt{\left(\frac{S_{open}}{k \cdot r \cdot \tan\frac{\pi}{k}}\right)^2 - 2\frac{S_{open}}{k \cdot \tan\frac{\pi}{k}} + \frac{r^2}{\cos^2\frac{\pi}{k}}} \\ &= \sqrt{\left(\frac{2k \cdot S_{open}}{k \cdot r \cdot \tan\frac{\pi}{k}}\right)^2 - 2\frac{4k^2 \cdot S_{open}}{k \cdot \tan\frac{\pi}{k}} + \frac{4k^2 \cdot r^2}{\cos^2\frac{\pi}{k}}} \\ &= \sqrt{\left(\frac{2 \cdot S_{open}}{r \cdot \tan\frac{\pi}{k}}\right)^2 - 8\frac{k \cdot S_{open}}{\tan\frac{\pi}{k}} + 4\frac{k^2 \cdot r^2}{\cos^2\frac{\pi}{k}}} \end{aligned} \tag{9}$$

The only case that yields a division by zero in Equation (9) happens when $k = 2$. That causes both $\tan\frac{\pi}{k}$ and $\cos\frac{\pi}{k}$ to return the value of ∞. In any other values of k, the functions are well-defined. Since we work on a discrete domain, the number of vertices the polygon that approximate the *open-space* area is bounded by the number of *open-space* points, which in our case equals to S_{open}:

$$k \leq S_{open} \tag{10}$$

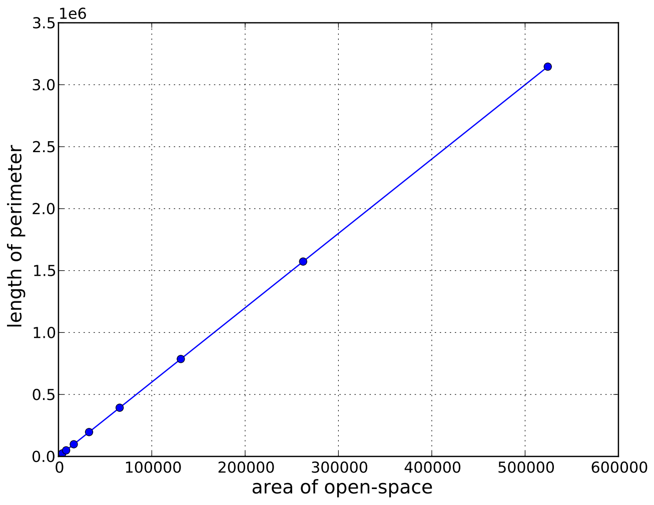


Figure 8: The linearity relation between the area of *open-space* region and its perimeter (Equation (9)). The x-axis contains samples of different *open-space* regions in sizes of: $2^2, 2^3, \ldots, 2^{20}$ and y-axis contains their perimeter size, respectively.

In the range $[3, \infty)$ the function $f(x) = \frac{1}{\cos x}$ is a monotonic deceasing function and therefore gets a maximal value of 2. This yields:

$$\frac{1}{\cos\frac{\pi}{k}} \leq 2 \qquad \text{for } k \in \mathbb{N} \setminus \{0, 1, 2\} \tag{11}$$

Therefore, Equations (9), (10) and (11) can be joint and we get Equation (12):

$$P_k \leq \sqrt{\left(\frac{2 \cdot S_{open}}{r \cdot \tan\frac{\pi}{S_{open}}}\right)^2 - 8\frac{S_{open}^2}{\tan\frac{\pi}{S_{open}}} + 2 \cdot S_{open}^2 \cdot r^2} \tag{12}$$

Therefore, when we join Equation (12) together with (1), we get the overall run-time complexity of *WFD* algorithm in terms of the *open-space* area, S_{open}:

$$\mathcal{O}\left(S_{open} + \sqrt{\left(\frac{2 \cdot S_{open}}{r \cdot \tan\frac{\pi}{S_{open}}}\right)^2 - 8\frac{S_{open}^2}{\tan\frac{\pi}{S_{open}}} + 2 \cdot S_{open}^2 \cdot r^2}\right)$$

After omitting constants:

$$\mathcal{O}\left(S_{open} + \sqrt{\left(\frac{S_{open}}{r \cdot \tan\frac{\pi}{S_{open}}}\right)^2 - \frac{S_{open}^2}{\tan\frac{\pi}{S_{open}}} + S_{open}^2 \cdot r^2}\right) \tag{13}$$

Figure 8 shows an evaluation of the *open-space* region's perimeter length as a function of the total area of *open-space* regions.

Figure 9 shows the graph for *WFD* final complexity function (Equation (13)), in the range of $S_{open} \in [0, 10000]$.

5.2 *FFD* Complexity Analysis

In Section 5.1, we showed that the run-time complexity of *WFD* is bounded by Equation (1). Allegedly, it may seem that *FFD*'s complexity is contained within *WFD* since *FFD* searches only inside the active area where all of its points are *open-space*. However, this is not true since *FFD* has to persistently run in the background. It may thus run a number of times by every single run of *WFD*. Moreover, it run for each particle, in particle-based mapping systems. As shown in Section 4, *FFD* contains four stages. We now analyse the complexity of each stage separately:

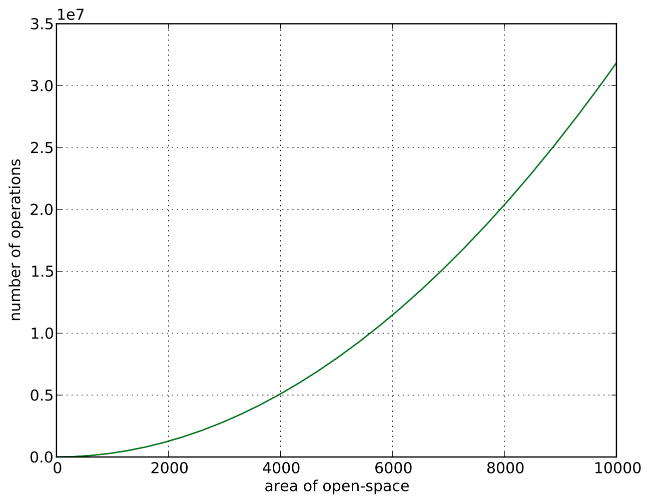


Figure 9: *WFD*'s complexity function, Equation (13)

5.2.1 Sorting stage (Algorithm 4.2)

FFD performs polar-sorting of laser readings. *FFD* utilizes cross-product instead of actually calculating angle and radius (which are relatively very time-consuming calculations). As explained in Section 4, in order to determine the order of the points received from the laser readings, we calculate the cross-product between them. Cross-product calculation is performed in a constant time and sorting is performed in time of $\mathcal{O}(n \log n)$. We denote l_r as the number of laser readings received in each measurement and therefore, total complexity of this stage is:

$$\mathcal{O}(l_r \log l_r) \cdot \mathcal{O}(1) = \mathcal{O}(l_r \log l_r) \tag{14}$$

5.2.2 Contour stage (Algorithm 4.3)

FFD scans each two adjacent points received from the polar-sorted laser readings. For each two adjacent points *FFD* calls *Bresenham's line algorithm* which returns a set of the points that lie between them. *Bresenham's line algorithm* is performed in linear time complexity. In the end, *FFD* merges all received lines into a single contour. We denote d_{p_i,p_j} as the euclidean distance between points p_i and p_j. Therefore, total complexity of this stage is the total distance between each two adjacent points, which is denoted as $c^{(t)}$, the length of the contour in time t:

$$\mathcal{O}\left(\sum_{p_i \in l_r} d_{p_{i-1},p_i}\right) = \mathcal{O}\left(c^{(t)}\right) \tag{15}$$

5.2.3 Detecting new frontiers stage (Algorithm 4.4)

FFD scans the contour from the previous stage and extracts new frontiers (if available of course). Each found new detected frontier is added into a list (which will be used in next stage) and hence, the addition is performed in a constant time. Lines 2–6 handle the special case when the first point on the contour is a frontier point. All actions are performed in a constant time. Therefore, total complexity of this stage is proportional to the length of the given contour:

$$\underbrace{\mathcal{O}(1)}_{\text{special case}} + \underbrace{\mathcal{O}\left(c^{(t)}\right) \cdot \mathcal{O}(1)}_{\text{general case}} = \mathcal{O}\left(c^{(t)}\right) \tag{16}$$

5.2.4 Maintenance stage (Algorithm 4.5)

Elimination of previous frontiers (Lines 2–11) *FFD* scans each point which lies inside the active area and checks if it was previously belonged to a frontier. Checking a specific point is performed in a constant time since all points in map already store a frontier index. The index is used as a lookup index in the frontier database (full implementation details can be found in Section 4). We denote $A^{(t)}$ as the bounding rectangle (the active area) in time t. Hence, scanning all points in the active area can be done in time of $\mathcal{O}\Big(A^{(t)}\Big) \cdot \mathcal{O}(1) = \mathcal{O}\Big(A^{(t)}\Big)$. We denote $f_{max}^{(t)}$ as the length of the longest frontier that exists in the frontier database in time t and $n_f^{(t)}$ as the number of frontiers that are stored in the frontier database in time t. If a previously frontier point is found, then *FFD* searches for its frontier in the database and removes this point from the frontier. Therefore, for a specific frontier point it takes $\mathcal{O}\Big(\log n_f^{(t)}\Big)$ to find the frontier in the frontier database by the lookup index (we used a self-balancing binary search tree in our implementation), $\mathcal{O}\Big(f_{max}^{(t)}\Big)$ to locate the specific point in the found frontier and $\mathcal{O}(1)$ to remove it from the found frontier. Therefore, total complexity for this stage is:

$$\mathcal{O}\Big(A^{(t)}\Big) \cdot \mathcal{O}\Big(f_{max}^{(t)} + \log n_f^{(t)}\Big) = \mathcal{O}\Big(A^{(t)} \cdot \Big(f_{max}^{(t)} + \log n_f^{(t)}\Big)\Big) \tag{17}$$

Adding new frontiers (Lines 12–20) *FFD* scans each frontier point within all new detected frontiers that were found at the detection stage. We denote $n_{new}^{(t)}$ as the number of new frontiers that were found in time t and hence, scanning all new detected frontiers that are found in time t is performed in complexity of $\mathcal{O}\Big(n_{new}^{(t)}\Big)$. Next, *FFD* checks if each new detected frontier point belongs to another previously detected frontier. As mentioned before, this check can be performed in a constant time since each point stores a frontier lookup index. Therefore, scanning all points in a specific frontier is performed in time of $\mathcal{O}\Big(f_{max}^{(t)}\Big)$ and searching the frontier whose index is stored within the point is performed in time of $\mathcal{O}\Big(\log n_f^{(t)}\Big)$. In the end, the new frontier is merged with the existing frontier which is performed in a constant time. Total complexity of this stage is:

$$\mathcal{O}\Big(n_{new}^{(t)}\Big) \cdot \mathcal{O}\Big(f_{max}^{(t)} + \log n_f^{(t)}\Big) = \mathcal{O}\Big(n_{new}^{(t)} \cdot \Big(f_{max}^{(t)} + \log n_f^{(t)}\Big)\Big) \tag{18}$$

5.2.5 Combining all stages

By joining together equations: (14), (15), (16), (17) and (18), we get the total complexity for one iteration:

$$\begin{aligned}
&\mathcal{O}\left(\underbrace{l_r \log l_r}_{\text{sorting}} + \underbrace{c^{(t)}}_{\text{contour}} + \underbrace{c^{(t)}}_{\text{detection}} + \overbrace{\underbrace{A^{(t)} \cdot \Big(f_{max}^{(t)} + \log n_f^{(t)}\Big)\Big)}_{\text{eliminate existing frontiers}} + \underbrace{n_{new}^{(t)} \cdot \Big(f_{max}^{(t)} + \log n_f^{(t)}\Big)}_{\text{add new frontiers}}}^{\text{frontiers maintenance}}\right) \\
= \; &\mathcal{O}\Big(l_r \log l_r + c^{(t)} + \Big(A^{(t)} + n_{new}^{(t)}\Big) \cdot \Big(f_{max}^{(t)} + \log n_f^{(t)}\Big)\Big)
\end{aligned} \tag{19}$$

All new detected frontiers are found only inside the active area. Thus, the number of new detected frontiers is bounded by size of the active area:

$$n_{new}^{(t)} \le A^{(t)} \tag{20}$$

All new detected frontiers lie within the contour. In the worst-case, all points that lie within the contour belong to the longest frontier. Hence, we can bound the length of the longest frontier by the contour's length:

$$f_{max}^{(t)} \le c^{(t)} \tag{21}$$

Therefore, by joining the upper bounds from equations (20) and (21) together with (19) we bound the complexity of *FFD* by Equation (22) below:

$$\begin{aligned}
&\mathcal{O}\Big(l_r \log l_r + c^{(t)} + \Big(A^{(t)} + A^{(t)}\Big) \cdot \Big(c^{(t)} + \log n_f^{(t)}\Big)\Big) \\
= \; &\mathcal{O}\Big(l_r \log l_r + c^{(t)} + A^{(t)} \cdot \Big(c^{(t)} + \log n_f^{(t)}\Big)\Big)
\end{aligned} \tag{22}$$

The size of the active area can be bounded by the maximum range of the laser. The largest active area rectangle is received only when the the laser scans an open area. In this case, the active area is a rectangle that bounds a circle whose radius equals to the maximum laser range:

$$A^{(t)} \le (2l_m) \times (2l_m) = 4(l_m)^2 \tag{23}$$

Joining together (22) and (23):

$$\begin{aligned} & \mathcal{O}\Big(l_r \log l_r + c^{(t)} + 4(l_m)^2 \cdot \Big(c^{(t)} + \log n_f^{(t)}\Big)\Big) \\ = \quad & \mathcal{O}\Big(l_r \log l_r + c^{(t)} + 4(l_m)^2 \cdot c^{(t)} + 4(l_m)^2 \cdot \log n_f^{(t)}\Big) \\ = \quad & \mathcal{O}\Big(l_r \log l_r + \big(1 + 4(l_m)^2\big) \cdot c^{(t)} + 4(l_m)^2 \cdot \log n_f^{(t)}\Big) \end{aligned} \tag{24}$$

The maximal laser range and the number of laser readings (l_m and l_r, respectively) do not change during the execution. Hence, they both can be considered as constants and therefore, can be removed from the asymptotic upper bound:

$$\mathcal{O}\Big(c^{(t)} + \log n_f^{(t)}\Big) \tag{25}$$

5.3 Summary

In this section, we showed running-time upper-bounds for both *WFD* and *FFD* algorithms (Section 5.1 and Section 5.2, respectively). Until now, we discussed *FFD*'s run-time complexity in a single execution. However, as said in Section 4, *FFD* has to persistently run in the background. More precisely, in order to maintain previously detected frontiers, *FFD* is executed when a new laser reading is received. Traditional frontier detectors are executed on demand, usually when a *map-event* has occurred. A *map-event* is an event which is raised by the SLAM implementation when a sufficient amount of new data was received in order to produce a new map. In order to compare *FFD* to other traditional frontier detectors, we define l_ω as the frequency of the laser sensor (e.g. how many laser readings are received in a time unit of 1 second). In addition, we denote t_m to be the worst-case elapsed time between two following map-events in the (i.e. in times t_i and t_{i+1}). Finally, we denote P_n as the number of particles in the SLAM implementation. In implementations containing only one map, such as EKF-based systems, the value of P_n is 1. The reason is that as said in Section 4, each particle has to maintain its own map and thus, run its own instance of *FFD*. Thus, Equation (25) now considers the frequency of the laser sensor and we get Equation (26):

$$\mathcal{O}\Big(P_n \cdot (t_m \cdot l_\omega) \cdot \Big(c^{(t)} + \log n_f^{(t)}\Big)\Big) \tag{26}$$

It might seem that in Equation (26) the value of $P_n \cdot (t_m \cdot l_\omega)$ is constant during all *FFD* executions and hence, redundant. However, in real-world situations, this value plays a major role in determining *FFD*'s run-time. For example: although lower values of l_ω decrease the number of *FFD* executions in the time between two following map-events, increasing the number of *FFD* instances (e.g. increasing the number of particles in a particle-filter based SLAM implementation) causes the total run-time becoming slower. The question remains open: how do P_n, t_m, and l_ω values affect *FFD*'s run-time in the real-world? The answer to this question is presented in Section 7.

6 About Maintenance and Improving *WFD*

The *WFD* algorithm does not separate old and new frontiers and therefore, has to detect them with each execution. Therefore, *WFD*'s search domain is bounded to the whole map. In contrast, *FFD* algorithm is able to detect new frontiers only. In order to enable *FFD* eliminating previously detected frontiers, it has to persistently perform a maintenance. The general maintenance algorithm is shown in Section 4 (Algorithm 4.5). In this section we discuss the general concepts of the maintenance process (Section 6.1) and present two algorithms which represent a hybrid version of both *WFD* and *FFD* (the former is discussed in Section 6.2 and the latter is discussed in Section 6.3).

6.1 General Maintenance Concepts

In Section 4.1.4, we showed and explained the maintenance stage of the *FFD* algorithm (Algorithm 4.5). This process can be generalized and applied to other frontier detection algorithms. By applying maintenance, we can separate between the two parts that any frontier detection needs to handle: detection of new frontiers and elimination of existing frontiers.

The presented maintenance algorithm (Algorithm 4.5) is not bounded to a specific frontier detection algorithm. The algorithm gets its inputs and acts independently. We encourage the reader to find more details in Section 4.1.4. By integrating the maintenance algorithm into *WFD*, the frontier search domain can now be reduced (similar to *FFD*).

For instance, we integrated the maintenance mechanism into *WFD* algorithm and created two advanced versions of it which are presented in the following sections. The first, an incremental version of *WFD* is called *WFD-INC*, and the second, a parallel version which is called *WFD-IP*.

6.2 Incremental Wavefront Frontier Detector

Our first *WFD* improvement is called *WFD-INC* (Algorithm 6.1). In this version of *WFD*, instead of searching the whole map for frontier points, the search domain contains only points that lie inside the active area. The idea is based on the proof in Section 4.2; the only region in the map that contains changes (i.e. contains new frontiers and contains frontiers that are needed to be eliminated) is the active area. We define the active area to be the bounding rectangle that is constructed using the farthest laser readings received from last executions of *WFD-INC*.

Algorithm 6.1 Incremental Wavefront Frontier Detector (WFD-INC)

```
Require: $queue_m$ // queue, used for detecting frontier points from a given map
Require: $pose$ // current global position of the robot
Require: $frontiersDB$ // data-structure that contains last known frontiers
 1: function WFD-INC($activeArea, Map, pose, frontiersDB$)
 2:     if best particle index was changed from last execution then
 3:         clear all data from $frontiersDB$
 4:         $activeArea \leftarrow Map$ // active area contains all map points
 5:     end if
 6:     $queue_m \leftarrow \emptyset$
 7:     $newFrontiers \leftarrow \emptyset$
 8:     ENQUEUE($queue_m$, $pose$)
 9:     mark $pose$ as "Map-Open-List"

10:     while $queue_m$ is not empty do
11:         $p \leftarrow$ DEQUEUE($queue_m$)
12:         if $p$ is marked as "Map-Close-List" then
13:             continue
14:         end if
15:         if $p$ is a frontier point then
16:             $foundFrontier \leftarrow$ EXTRACT-FRONTIER-2D($p$)
17:             add $foundFrontier$ to $newFrontiers$
18:             mark all points of $foundFrontier$ as "Map-Close-List"
19:         end if

20:         for all $v \in adj(p)$ do
21:             $tags \leftarrow$ {"Map-Open-List", "Map-Close-List"}
22:             $isNeighbor \leftarrow v$ has at least one "Map-Open-Space" neighbor
23:             if $v$ not marked as one of $tags$ and $isNeighbor$ and $v \in activeArea$ then
24:                 ENQUEUE($queue_m$,$v$)
25:                 mark $v$ as "Map-Open-List"
26:             end if
27:         end for

28:         mark $p$ as "Map-Close-List"
29:     end while
30:     MAINTAIN-FRONTIERS($newFrontiers, frontiersDB, activeArea$)
31: end function
```

Similar to State-of-the-art frontier detection algorithms, *WFD* has to detect all frontiers in each execution. *WFD* has better performance than State-of-the-art frontier detection algorithm since it has a smaller search domain. However, comparing to *FFD*, it is still limited. If there were not changes in map orientation, then all previous frontiers should keep their map coordinates.

The *WFD-INC* algorithm exploits this fact and scans only regions that might contain changes in frontiers (i.e. regions which were hit by the laser sensors). If a map orientation was changed (Lines 2–5), then all previous frontier data is cleared and the algorithm start detecting all frontiers in the same way as *WFD* algorithm. Another modification from original *WFD* algorithm is when checking the adjacent points of current scanned point (Line 23). In this version of *WFD*, only points that lie inside the active area are scanned. This reduces the size of the search domain. The last modification from the original *WFD* algorithm is that a maintenance has to be called, since *WFD-INC* does not search over the whole map (Line 30).

Both *WFD* (Algorithm 3.1) and *WFD-INC* (Algorithm 6.1) share many common code lines. We highlighted the differences between both algorithms in order to help the reader to notice the differences between them more easily.

6.3 Incremental-Parallel Wavefront Frontier Detector

Our second improvement of *WFD* is actually an improvement of *WFD-INC* and called *WFD-IP*. If changes in map orientations happen too often, then *WFD-INC*'s performance is the same as *WFD* (but now including an overhead of maintenance). Our goal is therefore to artificially reduce the number of changes in map orientation as much as possible. The solution is simple: *WFD-IP* keeps a separate instance of *WFD-INC* algorithm for each particle. Whenever *WFD-IP* is executed, it executes each instance of *WFD-INC* according to each particle's map data. Therefore, all instances of *WFD-INC* never reach lines 2–5 and keep frontier data between executions is guaranteed.

Algorithm 6.2 Incremental-Parallel Wavefront Frontier Detector (WFD-IP)

```
Require: $particles$ // data-structure that contains all particles of the SLAM implamentation
1: for all Particle $p \in particles$ do
2:     WFD-INC($p \rightarrow activeArea$, $p \rightarrow Map$, $p \rightarrow pose$, $p \rightarrow frontiersDB$)
3: end for
```

7 Experimental Results

In this section, we describe the different experiments that were conducted in order to evaluate and compare the algorithms described in Sections 3, 4 and 6. In Section 7.1 we describe the settings of the testing environment. In Section 7.2 we compare all discussed frontier detection algorithms. In Section 7.3 we examine *FFD* algorithm in a finer resolution and test different aspects that might affect its performance.

7.1 Background

We have fully implemented *WFD*, *WFD-INC*, *WFD-IP* and *FFD* and performed testings on data obtained from the Robotics Data Set Repository (Radish) [17]. Figure 10 shows the environments used for the evaluation. *WFD*, *WFD-INC*, *WFD-IP* and *FFD* were compared with a *SOTA* (state-of-the-art) frontier detection algorithm, due to Wurm[1] and Burgard [40, 39].

To evaluate the algorithms, we integrated them into a single-robot exploration system. The system is based on GMapping, an open-source SLAM implementation [13, 14]. We integrated our code into the *ScanMatcher* component which is contained inside *gsp thread* (Grid SLAM Processor). At the time that a new MapEvent is raised, all frontier detection algorithms are executed according to current world state. Execution times are measured by Linux system-call *getrusage*, which measures the CPU-process time.

We used several environments taken from Radish [17]:

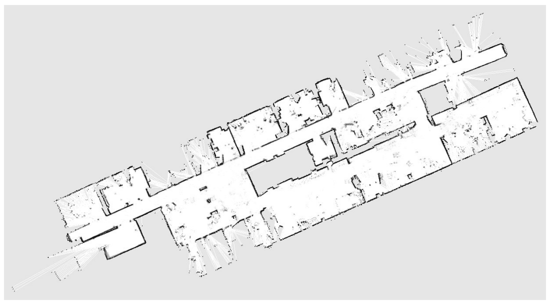
(A) Cartesium Building, University of Bremen

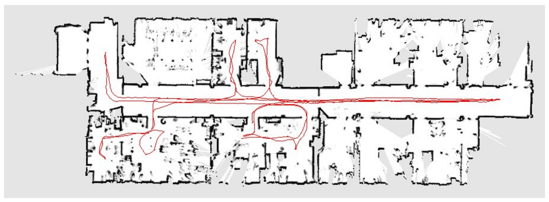
(B) Freiburg, Building 079

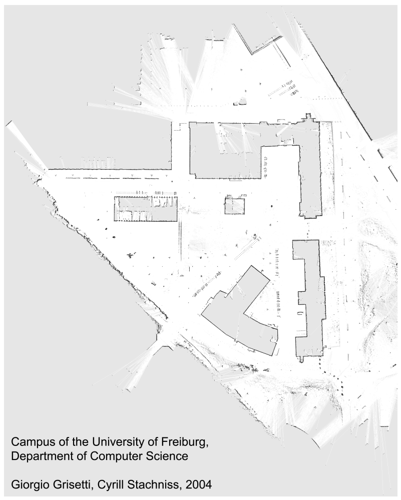
(C) Outdoor dataset recorded at the University of Freiburg

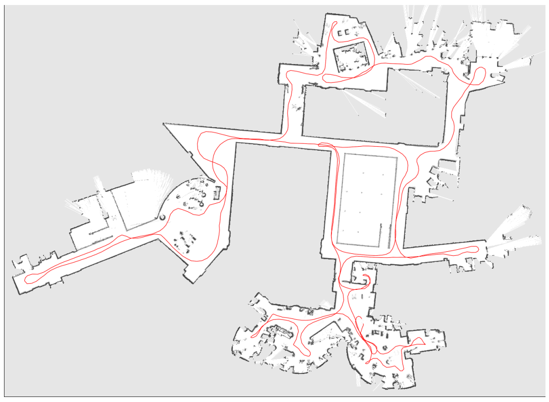
(D) 3rd Floor of MIT CSAIL

(E) Edmonton Convention Centre (site of the AAAI 2002 Grand Challenge)


(a) Cartesium Building, University of Bremen.


(b) Freiburg, Building 079.


Campus of the University of Freiburg,
Department of Computer Science
Giorgio Grisetti, Cyrill Stachniss, 2004
(c) Outdoor dataset recorded at the University of Freiburg


(d) 3rd Floor of MIT CSAIL

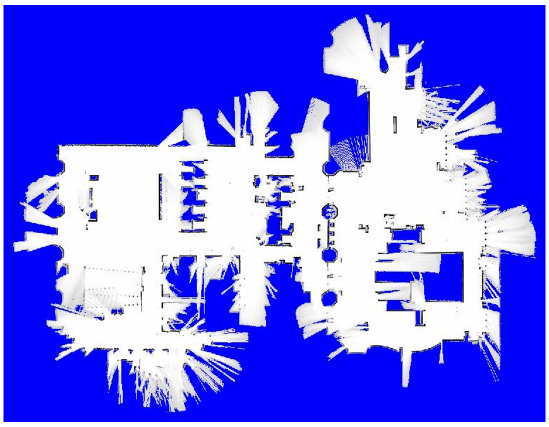

(e) Edmonton Convention Centre (site of the AAAI 2002 Grand Challenge)

Figure 10: Testing environments.

Note that we use the exploration data (raw sensor readings and odometry) from these data sets, and thus all algorithms use exactly the same data, form the same robot trajectories. Thus the movement of the robot is identical, and the only thing we examine is how quickly it can compute frontiers.

7.2 Comparing *SOTA*, *FFD*, *WFD*, *WFD-INC* and *WFD-IP*

We begin by examining overall performance. We examined the run-time of all algorithms on two different machines:

- First experiment: we used a fast desktop computer containing Intel Core 2 Duo T6600 CPU with clock speed of 2.20 GHz, Random Access Memory (RAM) in size of 4 GB, L1 cache in size of 64KB (8-way Set-associative) and L2 cache in size of 2048KB (8-way Set-associative).
- Second experiment: we used a slower desktop computer containing Intel Pentium III (Coppermine) with clock speed of 800 MHz, Random Access Memory (RAM) in size of 1 GB, L1 cache in size of 32KB (4-way Set-associative) and L2 cache in size of 256KB (4-way Set-associative). Research-grade robots typically have a faster CPU, but commercial robots typically do not.

FFD is called every-time a new laser reading is received. Therefore, in order to compare *FFD* execution time to other algorithms correctly, we accumulate *FFD*'s execution times between calls to other algorithms. In other words, if we call *WFD* in time-stamps t_i and t_{i+1}, then *FFD*'s accumulated execution time is calculated by:

$$\sum_{x=t_i}^{t_{i+1}} ExecutionTime_{FFD}(x)$$

Moreover, we remind the reader that because *FFD* is called for every particle in the particle-filtering GMapping [14], the results here accumulate also over the number of particles (30 in our case).

Figure 11 shows results of the comparison in each of the two machines. Each group of bars represents a run over a separate map. For each algorithm, we calculate the mean execution time, over the duration of the exploration. The vertical axis measures the calculated execution time in microseconds, on a *logarithmic scale*. The one-second line is at 10^6 microseconds. The next tick, at 10^7, marks 10 seconds.

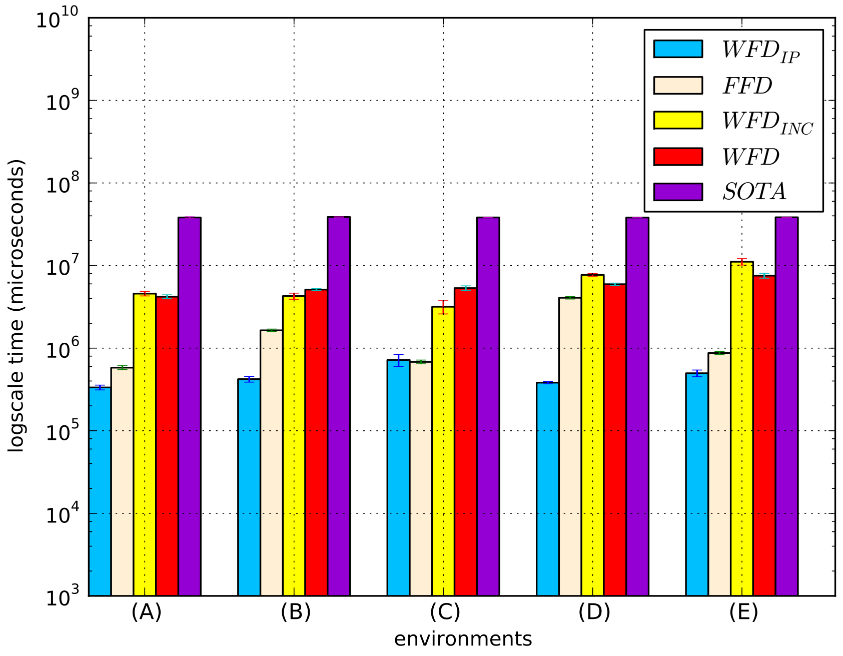
Figure 11 shows that *WFD* is faster than *SOTA* by approximately one order of magnitude. *FFD* is faster than *WFD* by one to two orders of magnitude. Indeed, *FFD* performs close to the one-second line. In contrast, *WFD* and *SOTA* typically take anywhere from 10 to 100 seconds to perform their task, even on relatively fast machines. Surprisingly, we discovered that *WFD-IP* often performs frontier detection in the shortest execution time. The reason is that *WFD-IP* contains a well balance between the frequency of number of calls (only on map-events). On the one hand, *WFD* algorithm has to detect all frontiers in each execution because it does not utilize previous detected frontiers. On the other hand, *FFD* algorithm keeps utilizing previous detected frontiers but in price of keep running the maintenance routine in the background. *WFD-IP* algorithm is an hybrid approach that reduces the number of calls (relatively to *FFD*) while performing frontier detection on a smaller search domain (relatively to *WFD*).

FFD's improvement over *SOTA* and *WFD* is indeed notable, given that the measured results are not for single *FFD* runs, but in fact show accumulated run-time, over the frequency of the sensor readings, multiplied over the number of particles (approximately 2000 calls to *FFD* for each *WFD* or *SOTA* calls). These multiplicative factors have significant impact on *FFD*'s usability. It is important to understand whether the number of particles influences the result more than the frequency of sensor readings, as the number of particles is often increased for better quality. We discuss this in detail in Section 7.3.

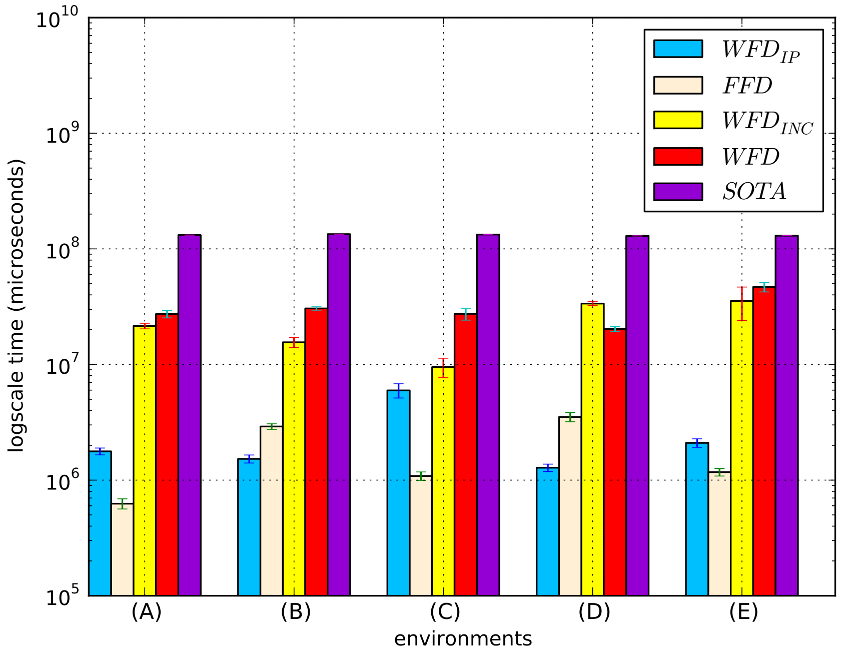
7.2.1 What Happened in Environment (D)?

If we take a closer look at the results shown in Figure 11(a), in environment (D) the mean run-time of *WFD* is relatively not much slower than *FFD*. Environment (D) contains plenty of regions that seems to have small obstacles which enlarge the length of the contour scanned by *FFD*. In Section 5.2 we showed an upper bound of *FFD*'s run-time complexity (Equation (26)). One of the factors that affect the boundary is the contour length that is scanned by *FFD* algorithm (Section 4.1.3).

[1]We thank Kai M. Wurm and Wolfram Burgard for providing us with their own implementation of state-of-the-art frontier detection algorithm.



(a) Intel T6600



(b) Intel Coppermine

Figure 11: Comparing all versions of *WFD* and *FFD* to State-of-the-Art algorithm on different machines.

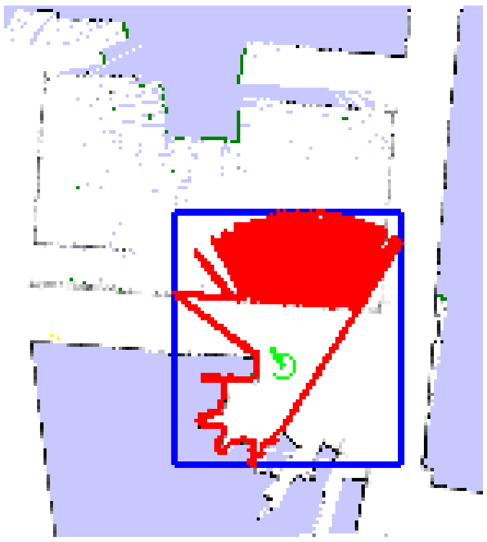


Figure 12: An example of *FFD*'s worst-case as shown in Environment (D)

Map	Changes	Executions	Percent (%)
(A)	56	110	50.91
(B)	52	287	18.12
(C)	25	160	15.62
(D)	193	429	44.99
(E)	73	340	21.47

Table 1: Comparing active particle changes in different environments.

Figure 12 shows an example of the enlargement of the contour when the robot arrives to a crowded region. Thus, The empirical results shown in this section well-support the theoretical boundary.

7.2.2 Why is *WFD-INC* Sometimes Worse than *WFD*?

In the worst-case, *WFD-INC* algorithm should perform the same as *WFD* including an overhead of maintenance. However, as can be seen in Figure 11, the run-time means of *WFD* and *WFD-INC* do not have a specific trend.

We hypothesize that in certain environments, where there is a high frequency of active particle change, *WFD-INC* cleans its previous detected frontiers more often, and this explains its run-time.

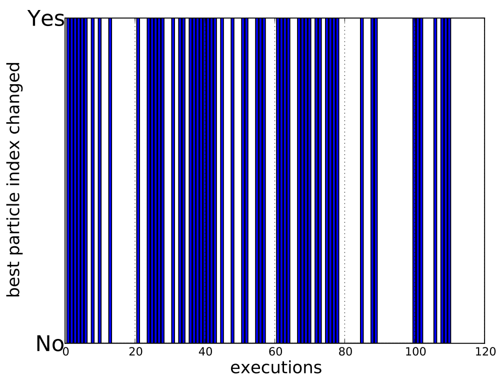
We provide evidence to support this hypothesis in Table 1 and Figure 13. Table 1 shows the particle changes in different environments. Each row represents a single environment. The first column contains the environment identifier. The second column contains the number of active particle changes. The third column contains the total number of executions and the forth column contains the percentage of particle changes relative to the total number of executions.

Figure 13 shows the active particle changes over time. For a specific environment, the vertical axis represents each execution and the horizontal axis represents whether the active particle was changed (*Yes* for a change and *No* otherwise). There is an exception in environment (C) in which the active particle changes percent is quite low but *WFD-INC*'s execution time is still worse than *WFD*'s execution time. Here, the explored environment is very large and hence, for each change, the robot has to detect frontiers in a very large area, in contrast to other environments.

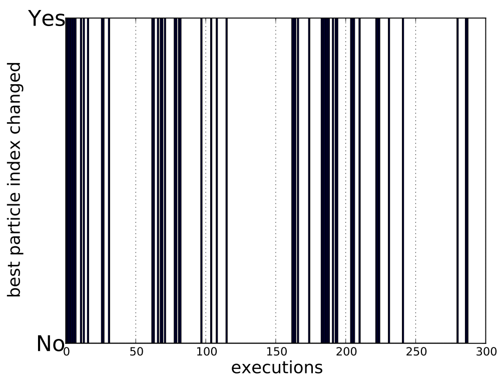
7.2.3 What Happened to *WFD-IP* in Environments (A),(C),(E), Figure 11(b)?

Figure 11 shows the run-time results of all algorithms on two testing machines. On the stronger machine (Figure 11(a)) *WFD-IP* outperforms all other algorithms. However, on the weaker machine (Figure 11(b)), the situation is different in environments (A),(C) and (E).

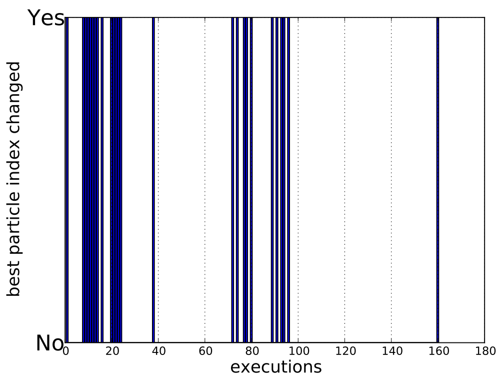
The two machines differ in both memory size and number of cores. The Coppermine machine is equipped with less random access memory (1 GB) than the stronger machine (4 GB). The key feature



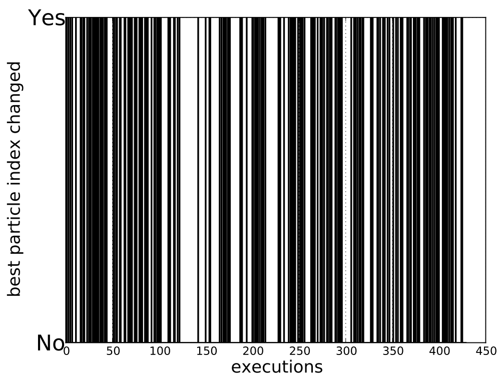
(a) Cartesium Building, Bremen



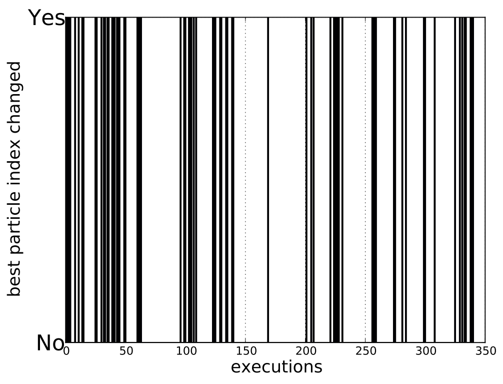
(b) Freiburg, Building 079



(c) Outdoor dataset, University of Freiburg



(d) 3rd Floor of MIT CSAIL



(e) Edmonton Convention Centre

Figure 13: Changes in Active Particle Over Executions.

of *WFD-IP* is its robustness against changes in active particle since it holds a separate instance of *WFD-INC* for each particle. In our implementation, there are 30 particles and hence, *WFD-IP* holds 30 instances of *WFD-INC* (i.e., 30 frontier databases, 30 maps etc.). In addition, the Coppermine machine is equipped with a single core CPU, in contrast to the stronger machine that is equipped with 2 cores. The GMapping SLAM system runs two threads (a *main* thread and a *SLAM* thread). We believe that the stronger machine allows each thread to run on a separate core.

There thus remains a question as to the reason for such behavior of *WFD-IP* relates to either the lack of memory or to the single core of the weaker machine. To determine this, we bounded the memory of the strong machine to be 1GB. The behavior of *WFD-IP* remained the same (we forgo adding the redundant figure here). The conclusion is therefore that the number of cores caused the differences in *WFD-IP*'s behavior. This should be taken into account when choosing between *FFD* and *WFD-IP* for a given robot CPU.

7.3 Evaluating *FFD* at a Finer Resolution

We turn to evaluating *FFD* more closely. Figure 14 compares the run-time of individual particles in specific environments. Each bar represents a specific particle. The vertical axis measures the mean run-time of *FFD* for the particle. The error-bars represent the standard deviation of each particle's run-time.

The figure shows that the per-particle run-time is measured in a few hundred micro-seconds. Thus the overall results were accumulating comparing the accumulation of thousands of *FFD* runs against single *WFD* and *SOTA* runs. Indeed, one can boost *FFD*'s execution time by not executing it on every received laser reading, since the frequency of receiving new laser readings is often higher than the speed of processing and updating the map anyways. Many laser sensors generate output at 30Hz–75Hz, at least three times faster than the rate at which the robots process the information. By ignoring some laser readings, *FFD* would perform much better, without any noticeable decay in mapping quality.

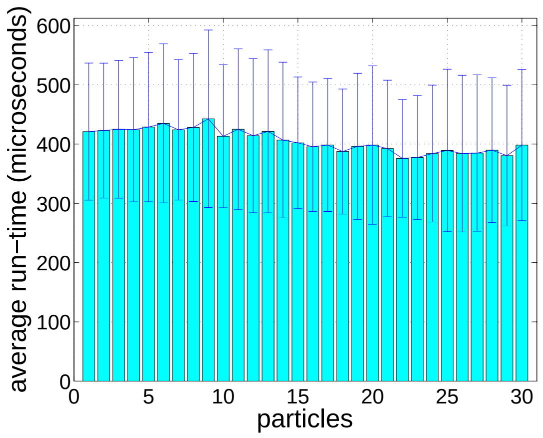
FFD's Performance According to Number of Particles One of *FFD*'s drawbacks is that in order to get a complete picture of frontiers in a given time, it has to persistently run in the background. As mentioned in Section 4.1.4, if *FFD* is integrated into a particle-filter based SLAM implementation, each particle has its own instance of *FFD* algorithm and hence, the overall run-time is increased. Figure 15 compares the mean run-time of *FFD* by changing the number of particles in specific environments. Each bar represents a run of *FFD* with a specific number of particles. The vertical axis measures the mean run-time of *FFD* for the configuration. The error-bars represent the standard deviation of each configuration's run-time.

Speeding-Up _WFD_ Even Further *WFD*'s execution time can be boosted even more by reducing the grid size (i.e., by using a coarse grid). Of course, there is a trade-off between shorter execution time and the quality of the output frontiers. Even though, standard exploration tasks can utilize the output frontiers received in this manner. The grid is divided into blocks in size of the robot's width and height. Smaller blocks will not make sure that robot will be able to pass through terrain obstacles (i.e. corridors). Each block in the real world is represented by a single cell in the reduced grid. In order to determine the occupancy value of the cell, we examined different strategies. We considered both the speed of creating the new grid and the quality of the output. We found out that sampling the center of the block edges and the block center yields the best results. We plan to investigate efficient techniques to reduce the grid size while considering the quality of the output coarse grid data.

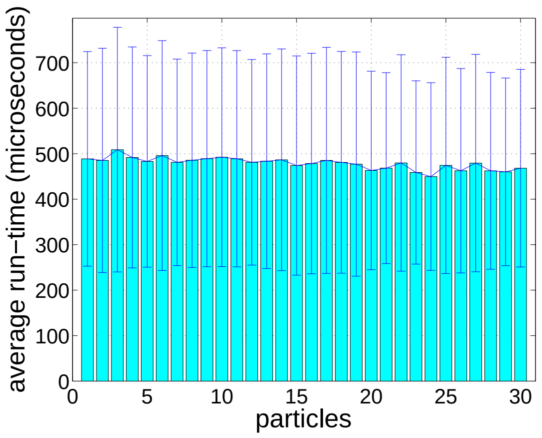
8 Conclusions and Future Work

Frontier-based exploration is the most common approach to solve the exploration problem. State-of-the-art frontier detection methods process the entire map data which hangs the exploration system for a few seconds with every call to the frontier detection algorithm.

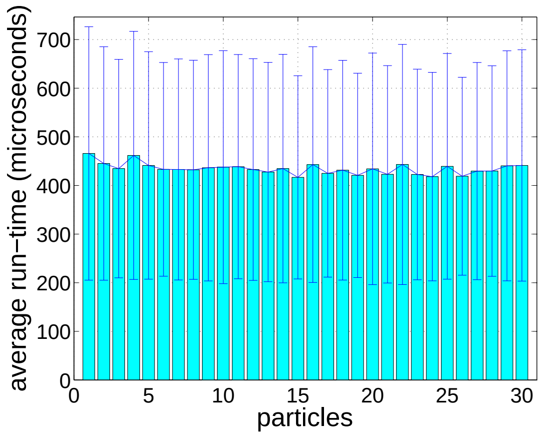
In this work we present four novel faster frontier detectors, *WFD*, *FFD*, *WFD-INC* and *WFD-IP*. The first algorithm, a graph-based search, processes the map points which have already been scanned by the robot sensors and therefore, does not process unknown regions in each run (though it grows slower as more area is known). The second algorithm, a laser-based approach for frontier detection, only processes new laser readings which are received in real time, eliminating also much of the known search area. However, maintaining previous frontiers knowledge requires tight integration



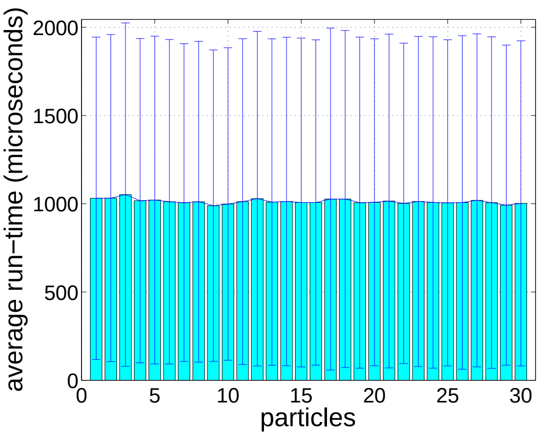
(a) Cartesium Building, Bremen



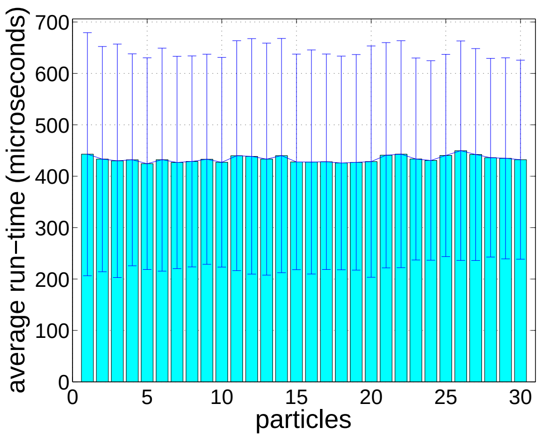
(b) Freiburg, Building 079



(c) Outdoor dataset, University of Freiburg

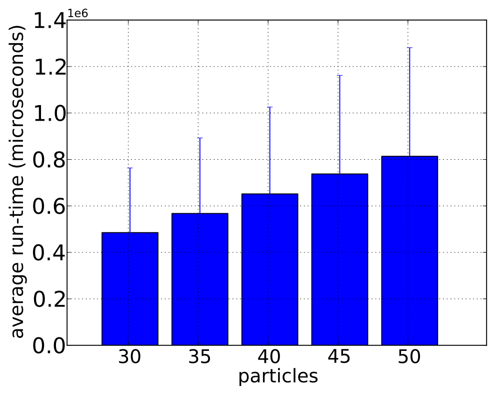


(d) 3rd Floor of MIT CSAIL

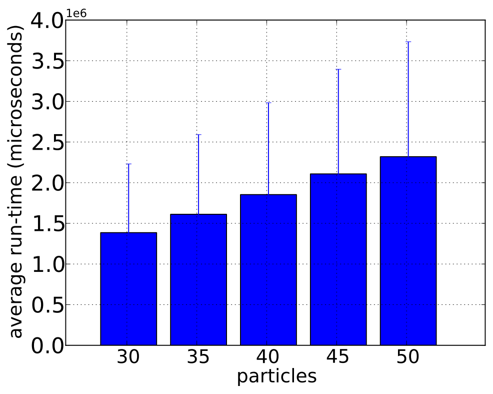


(e) Edmonton Convention Centre

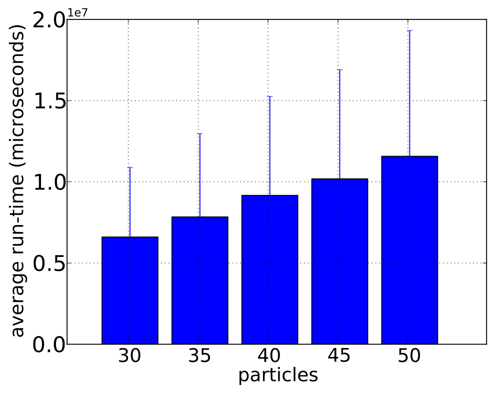
Figure 14: *FFD* run-time for individual SLAM particles.



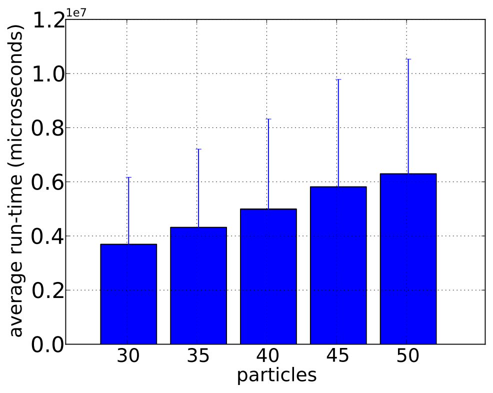
(a) Cartesium Building, Bremen



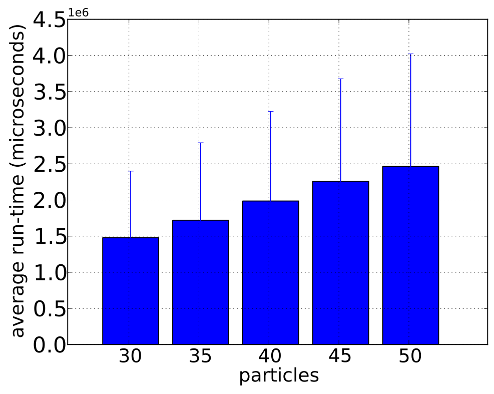
(b) Freiburg, Building 079



(c) Outdoor dataset, University of Freiburg



(d) 3rd Floor of MIT CSAIL



(e) Edmonton Convention Centre

Figure 15: *FFD*'s run-time according to number of particles.

with the mapping component, which may not be straightforward. The third and fourth algorithms are a combined approach of both *WFD* and *FFD*. Both algorithms search for frontiers within the known regions but their search space is still smaller than *WFD*'s search space since they search for frontiers only in the regions that were covered by the robot sensors since their last execution.

We describe efficient implementation for all algorithms and compare them empirically. *FFD* and *WFD-IP* are shown to outperform *WFD*, *WFD-INC* by 1–2 orders of magnitude. In addition, *FFD* and *WFD-IP* outperform state-of-the-art by 2–3 orders of magnitude.

In the future, we plan to integrate the general maintenance mechanism with *EKF*-based SLAM implementations, which we hope will lead to further improvements. We also plan to begin investigation of novel exploration policies, based on real-time frontier-detection.

References

[1] D. Apostolopoulos, L. Pedersen, B. Shamah, K. Shillcutt, M. Wagner, and W. Whittaker. Robotic antarctic meteorite search: Outcomes. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-01)*, pages 4174–4179, 2001.

[2] M. Batalin and G. Sukhatme. Efficient exploration without localization. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-03)*, volume 2, pages 2714–2719. IEEE, 2003.

[3] M. Berhault, H. Huang, P. Keskinocak, S. Koenig, W. Elmaghraby, P. Griffin, and A. Kleywegt. Robot exploration with combinatorial auctions. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03)*, pages 1957–1962, 2003.

[4] N. Bouraqadi, A. Doniec, and E. M. de Douai. Flocking-Based Multi-Robot Exploration. In *National Conference on Control Architectures of Robots*, 2009.

[5] J. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 2010.

[6] W. Burgard, M. Moors, D. Fox, R. Simmons, and S. Thrun. Collaborative multi-robot exploration. In *IEEE International Conference on Robotics and Automation. Vol. 1*, pages 476–481, 2000.

[7] W. Burgard, M. Moors, C. Stachniss, and F. Schneider. Coordinated multi-robot exploration. *IEEE Transactions on Robotics*, 21(3):376–378, 2005.

[8] D. Calisi, A. Farinelli, L. Iocchi, and D. Nardi. Multi-objective exploration and search for autonomous rescue robots. *Journal of Field Robotics*, 24(8-9):763–777, 2007.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[10] H. Durrant-Whyte and T. Bailey. Simultaneous localisation and mapping (SLAM): Part I the essential algorithms. *Robotics and Automation Magazine*, 13(99):80, 2006.

[11] D. Fox, J. Ko, K. Konolige, B. Limketka, D. Schulz, and B. Stewart. Distributed Multi-robot Exploration and Mapping. *Proceedings of the IEEE*, 94(1):1325–1339, 2010.

[12] C. Geyer. Practical markov chain monte carlo. *Statistical Science*, 7(4):473–483, 1992.

[13] G. Grisetti, C. Stachniss, and W. Burgard. Improving grid-based SLAM with Rao-Blackwellized particle filters by adaptive proposals and selective resampling. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-05)*, pages 2443–2448, 2005.

[14] G. Grisetti, C. Stachniss, and W. Burgard. Improved techniques for grid mapping with Rao-Blackwellized particle filters. *IEEE Transactions on Robotics*, 23:34–46, 2007.

[15] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[16] D. F. Hougen, S. Benjaafar, J. Bonney, J. Budenske, M. Dvorak, M. L. Gini, H. French, D. G. Krantz, P. Y. Li, F. Malver, B. J. Nelson, N. Papanikolopoulos, P. E. Rybski, S. Stoeter, R. M. Voyles, and K. B. Yesin. A miniature robotic system for reconnaissance and surveillance. In *Proceedings of IEEE International Conference on robotics and automation (ICRA-00)*, pages 501–507, 2000.

[17] A. Howard and N. Roy. The robotics data set repository (RADISH), 2003.

[18] M. Juliá, A. Gil, and O. Reinoso. A comparison of path planning strategies for autonomous exploration and mapping of unknown environments. *Autonomous Robots*, pages 1–18, 2012.

[19] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou, and S. Shimada. Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics (SMC-99)*, pages 739–746. IEEE Computer Society, 1999.

[20] J. Ko, B. Stewart, D. Fox, K. Konolige, and B. Limketkai. A practical, decision-theoretic approach to multi-robot mapping and exploration. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03)*, pages 3232–3238, 2003.

[21] K. Konolige, D. Fox, C. Ortiz, A. Agno, M. Eriksen, B. Limketkai, J. Ko, B. Morisset, D. Schulz, B. Stewart, et al. Centibots: Very large scale distributed robotic teams. *Experimental Robotics IX*, pages 131–140, 2006.

[22] K. Konolige, C. Ortiz, R. Vincent, A. Agno, M. Eriksen, B. Limketkai, M. Lewis, L. Briesemeister, E. Ruspini, D. Fox, et al. Large scale robot teams. *Multi-Robot Systems: From Swarms to Intelligent Autonoma*, 2:193–204, 2003.

[23] B. Kuipers and Y. Byun. A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations. *Robotics and autonomous systems*, 8(1-2):47–63, 1991.

[24] H. Lau. Behavioural approach for multi-robot exploration. In *Australasian Conference on Robotics and Automation (ACRA)*, Brisbane, December 2003.

[25] J. Leonard and H. Durrant-Whyte. Mobile robot localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation*, 7(3):376–382, 1991.

[26] R. M. Neal. Probabilistic inference using markov chain monte carlo methods. Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto, September 1993.

[27] D. Puig Valls. Balanced multi-robot exploration through a global optimization strategy. *Journal of Physical Agents*, 4(1):35–44, 2010.

[28] I. Rekleitis, G. Dudek, and E. Milios. Multi-robot exploration of an unknown environment, efficiently reducing the odometry error. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 15, pages 1340–1345, 1997.

[29] I. Rekleitis, G. Dudek, and E. Milios. Multi-robot collaboration for robust exploration. *Annals of Mathematics and Artificial Intelligence*, 31(1):7–40, 2001.

[30] R. Sawhney, K. M. Krishna, and K. Srinathan. On fast exploration in 2D and 3D terrains with multiple robots. In *Proceedings of the Eighth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-09)*, pages 73–80, 2009.

[31] R. Simmons, D. Apfelbaum, W. Burgard, D. Fox, M. Moors, S. Thrun, and H. Younes. Coordination for multi-robot exploration and mapping. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pages 852–858, 2000.

[32] C. Stachniss. *Exploration and Mapping with Mobile Robots*. PhD thesis, University of Freiburg, Department of Computer Science, 2006.

[33] C. Stachniss, O. Mozos, and W. Burgard. Speeding-up multi-robot exploration by considering semantic place information. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-06)*, pages 1692–1697. IEEE, 2006.

[34] S. Stoeter, P. Rybski, M. Erickson, M. Gini, D. Hougen, D. Krantz, N. Papanikolopoulos, and M. Wyman. A robot team for exploration and surveillance: design and architecture. In *Proceedings of the International Conference on Intelligent Autonomous Systems*, pages 767–774, 2000.

[35] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. Intelligent robotics and autonomous agents series. The MIT Press, August 2005.

[36] A. Visser. personal communication. Email, January 4th, 2011.

[37] A. Visser and B. A. Slamet. Including communication success in the estimation of information gain for multi-robot exploration. In *Proceedings of the 6th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt-08)*, pages 680–687, 2008.

[38] G. Welch and G. Bishop. An introduction to the kalman filter. *University of North Carolina, Chapel Hill, NC*, 7(1), 1995.

[39] K. Wurm. personal communication. Email, January 20th, 2011.

[40] K. Wurm, C. Stachniss, and W. Burgard. Coordinated multi-robot exploration using a segmentation of the environment. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-08)*, Nice, France, Sept. 2008.

[41] B. Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA-97)*, pages 146–151, Washington, DC, USA, 1997. IEEE Computer Society.

[42] B. Yamauchi. Frontier-based exploration using multiple robots. In *Proceedings of the Second International Conference on Autonomous Agents (Agents-98)*, pages 47–53, 1998.

[43] R. Zlot, A. Stentz, M. Dias, and S. Thayer. Multi-robot exploration controlled by a market economy. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-02)*, volume 3, pages 3016–3023. IEEE, 2002.