Robot Exploration with Fast Frontier Detection: Theory and Experiments

Matan Keidar
MAVERICK Group, Department of Computer Science, Bar-Ilan University
matankdr@gmail.com

Gal A. Kaminka
MAVERICK Group, Department of Computer Science, Bar-Ilan University
galk@cs.biu.ac.il

ABSTRACT

Frontier-based exploration is the most common approach to exploration, a fundamental problem in robotics. In frontier-based exploration, robots explore by repeatedly computing (and moving towards) *frontiers*, the segments which separate the known regions from those unknown. However, most frontier detection algorithms process the entire map data. This can be a time consuming process which slows down the exploration. In this paper, we present two novel frontier detection algorithms: *WFD*, a graph search based algorithm and *FFD* , which is based on processing only the new laser readings data. In contrast to state-of-the-art methods, both algorithms do not process the entire map data. We implemented both algorithms and showed that both are faster than a state-of-the-art frontier detector implementation (by several orders of magnitude).

General Terms

Algorithms Performance

Keywords

Robot, Exploration, Frontier, Laser

1. INTRODUCTION

The problem of exploring an unknown territory is a fundamental problem in robotics. The goal of exploration is to gain as much new information as possible of the environment within bounded time. 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 [17] was the first to show a frontier-based exploration strategy. His work preceded many others (e.g, [3, 4, 10, 11]).

Most frontier detection methods are based on edge detection and region extraction techniques from computer vision. To detect frontiers, they process the entire map data with every execution to the algorithm. State-of-the-art frontier detection algorithms can take a number of seconds to run, even on powerful computers. If a large region is explored, the robot actually has to wait in its spot until the frontier detection algorithm terminates. Therefore, many exploration implementations call the frontier detection algorithm only when the robot arrives at its destination.

This 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, are exploring the environment, from their initial locations (Figure 2(a)). One of the robots passes by a target assigned to the other, thus clearing it (Figure 2(b)). But because the other robot cannot continuously re-detect frontiers, it unnecessarily continues towards the covered target, instead of turning to more fruitful exploration targets.

In this paper, we thus focus on significantly speeding up frontier detection. We introduce two 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. It builds on ideas suggested in earlier work [5] which were not evaluated as an alternative to the edge-detection state-of-the-art. The key idea in *WFD* is that it does not scan the entire map, only the regions that have already been visited by the robot. 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

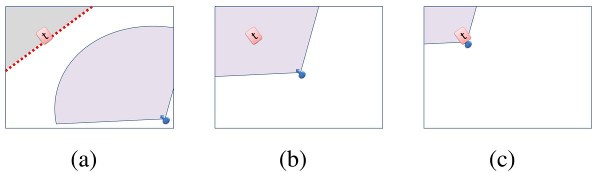


Figure 1: A single-robot example. In 1(a) the robot is heading towards the marked target on the frontier. In 1(b) the target and all of the remaining are covered by the robot's sensors, but because the robot does not re-detect frontiers, it continues to move. In 1(c) the robot has reached the frontier,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

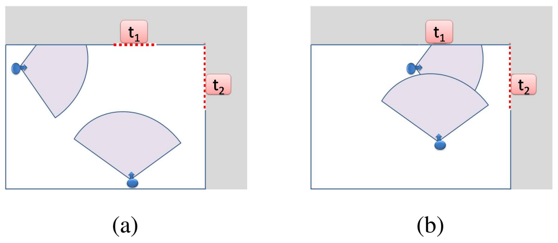


Figure 2: A multi-robot example. In 2(a), the top robot (R_2) is heading towards the right target, t_2; the other robot (R_1) heads towards the top target t_1. In 2(b) R_2 has reached its target, clearing both t_1 and t_2, making R_1's movements unnecessary.

Appears in: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), 4-8 June 2012, Valencia, Spain.

sensor data, it requires extending the mapper (SLAM) 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.

We provide a detailed evaluation of these algorithms, and contrast them with the state-of-the-art (SOTA). We examine their performance in different types of environments and two different CPUs. We show that *WFD* is faster than SOTA by 1–2 orders of magnitude, and that *FFD* is faster than WFD by 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 methods which were impractical until now.

2. RELATED WORK

An outline of the exploration process can be described as follows: while there is an unknown territory, allocate each robot a target 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. To the best of our knowledge, all of the following works utilize a standard edge-detection method for computing the frontiers. They recompute frontier 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 [17] 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. [3, 4] focus their investigation on 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. [15] 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. Wurm [16] suggests to call frontier detection every time-step of the coordination algorithm. Moreover, he claims that updating frontiers frequently is important in a multi-robot team since the map is updated not only by the robot assigned to a given frontier but also by all of the robots in the team. He suggests executing the algorithm $0.5m - 1m$ or every second or whenever a new target is requested.

Stachniss [12] 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 new targets are needed 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. [1] 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. [14] investigated how limited communication range affects multi-robot exploration. They proposed an algorithm which takes into account wireless constraints when selecting frontier targets. Visser [13] suggests recomputing frontiers every 3–4 meters, which in his opinion, has positive effect.

Our work on *WFD* is independent from previous work, though [5] mentions a frontier detection algorithm that utilizes breadth-first search, similar to *WFD* . However, [5] does not provide details of the algorithm, nor evaluation of its performance, and so exact similarities and differences cannot be assessed. Our work here also significantly extends and corrects our own earlier work [9], which presented preliminary—and incomplete—versions of the *WFD* and *FFD* algorithms. Compared to [9], this paper presents corrected algorithms, proves the soundness and completeness of *FFD* , and reports new experimental and analytical results.

3. WAVEFRONT FRONTIER DETECTOR

We present a graph search based approach for frontier detection. The algorithm, *WFD* (Algorithm 1), 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.

3.1 Definitions and Terms

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 WFD

WFD (Algorithm 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 (Lines 1–3).

Next, a BFS is performed (Line 4–30) in order to find all frontier points contained in the map. The algorithm keeps scanning only

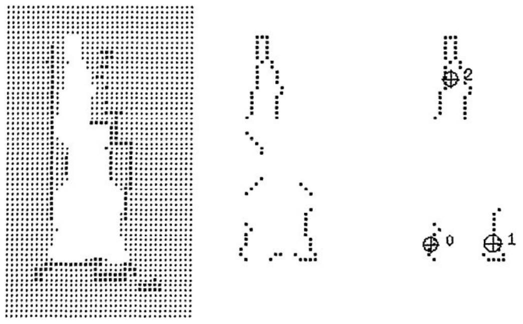


Figure 3: Evidence grid, frontier points, extraction of different frontiers (from left to right). Taken from [17].

Algorithm 1 Wavefront Frontier Detector (WFD)

```
Require: $queue_m$ // queue, used for detecting frontier points from a given map
Require: $queue_f$ // queue, used for extracting a frontier from a given frontier cell
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:     if $p$ is a frontier point then
9:         $queue_f \leftarrow \emptyset$
10:        $NewFrontier \leftarrow \emptyset$
11:        ENQUEUE($queue_f, p$)
12:        mark $p$ as "Frontier-Open-List"

13:        while $queue_f$ is not empty do
14:            $q \leftarrow$ DEQUEUE($queue_f$)
15:            if $q$ is marked as {"Map-Close-List","Frontier-Close-List"} then
16:                continue
17:            if $q$ is a frontier point then
18:                add $q$ to $NewFrontier$
19:                for all $w \in adj(q)$ do
20:                    if $w$ not marked as {"Frontier-Open-List","Frontier-Close-List", "Map-Close-List"} then
21:                        ENQUEUE($queue_f$,$w$)
22:                        mark $w$ as "Frontier-Open-List"
23:            mark $q$ as "Frontier-Close-List"
24:        save data of $NewFrontier$
25:        mark all points of $NewFrontier$ as "Map-Close-List"
26:    for all $v \in adj(p)$ do
27:        if $v$ not marked as {"Map-Open-List","Map-Close-List"} and $v$ has at least one "Map-Open-Space" neighbor then
28:            ENQUEUE($queue_m$,$v$)
29:            mark $v$ as "Map-Open-List"
30:    mark $p$ as "Map-Close-List"
```

points that have not been scanned yet and represent open-space (Line 27). 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 (Line 30). If a frontier point is found, a new BFS is performed in order to extract its frontier (Lines 13–25). This BFS searches for frontier points only. Extracting the frontier is ensured because of the connectivity of frontier points. At the end of the extraction (Line 25), the extracted frontier data is saved to a set data-structure that stores all frontiers found in the algorithm 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 (Line 28)
2. Map-Close-List: points that have already been dequeued by the outermost BFS (Line 5)
3. Frontier-Open-List: points that have already been enqueued by the frontier extraction BFS (Line 21)
4. Frontier-Close-List: points that have already been dequeued by the frontier extraction BFS (Line 14)

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.

3.3 Speeding-Up WFD Even Further

WFD's execution time can be boosted even more by reducing the grid size. 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 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.

4. FAST FRONTIER DETECTOR

Unlike other frontier detection methods (including *WFD*), our proposed algorithm (Algorithm 2) 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 and not time-efficient. The *FFD* algorithm contains four steps (Algorithm 2), and can be called with every new scan.

4.1 Sorting

The first step (line 1) 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.

To sort the readings, we assume that range readings are a set of Cartesian coordinated points, which consists of the locations

Algorithm 2 Fast Frontier Detector (FFD)

```
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

    // polar sort readings according to robot position
 1: $sorted \leftarrow SORT\_POLAR(lr, pose)$
    // get the contour from laser readings
 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:         $contour \leftarrow contour \cup \{p\}$
    // extract new frontiers from contour
 8: $NewFrontiers \leftarrow \emptyset$ // list of new extracted frontiers
 9: $prev \leftarrow POP(contour)$
10: if $prev$ is a frontier cell then // special case
11:     create a new frontier in $NewFrontiers$
12: for all Point $curr \in contour$ do
13:     if $curr$ is not a frontier cell then
14:         $prev \leftarrow curr$
15:     else if $curr$ has been visited before then
16:         $prev \leftarrow curr$
17:     else if $curr$ and $prev$ are frontier cells then
18:         add $curr$ to last created frontier
19:         $prev \leftarrow curr$
20:     else
21:         create a new frontier in $NewFrontiers$
22:         add $curr$ to last created frontier
23:         $prev \leftarrow curr$
    // maintainance of previously detected frontiers
24: for all Point $p \in ActiveArea$ do
25:     if $p$ is a frontier cell then
26:         // split the current frontier into two partial frontiers
27:         get the frontier $f \in frontiersDB$ which enables $p \in f$
28:         $f_1 \leftarrow f[1 \ldots p]$
29:         $f_2 \leftarrow f[(p+1) \ldots |f|]$
30:         remove $f$ from $frontiersDB$
31:         add $f1$ and $f2$ to $frontiersDB$
32: for all Frontier $f \in NewFrontiers$ do
33:     if $f$ overlaps with an existing frontier $existFrontier$ then
34:         $merged \leftarrow f \cup existFrontier$
35:         remove $existFrontier$ from $frontiersDB$
36:         add $merged$ to $frontiersDB$
37:     else
38:         create a new index and add $f$ to $frontiersDB$
```

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 [6] 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 $\overrightarrow{P_0 P_1}$ is clockwise from $\overrightarrow{P_0 P_2}$. 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*.

4.2 Contour

In this step (lines 2–7) 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* [2]. 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).

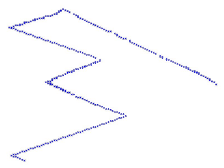


Figure 4: Example of produced contour.

4.3 Detecting New Frontiers

In this step (lines 8–23) 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.
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.
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.

4.4 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 (lines 24–38), 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 targets: avoiding detection of new frontiers in an already scanned area (Section 4.4.2), eliminating frontier points which are no longer belong to frontiers (Section 4.4.3) and joining correctly the new detected frontiers together with previously detected frontiers (Section 4.4.4).

4.4.1 *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 paper), 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.

Grid of Frontier Indices This data-structure 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 is into the Frontier Database (described below). In our implementation, we used integer index values. After accessing a grid cell, querying for its frontier index is $O(1)$.

Frontier Database This data-structure maps frontier indexes (pointers) to sets of points. 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.

4.4.2 *Avoiding Re-Detection of Same Frontier*

FFD detects new frontiers by processing laser readings only. Hence, *FFD* might detect the same frontier again and classify it as a new frontier if the robot did not change its position during two following *FFD* executions. Moreover, if the robot travels back to an already visited region, no new frontiers should be detected. *FFD* has to distinguish between laser readings from between time frames. Otherwise, *FFD* might wrongly detect a new frontier which lies within an already scanned area.

Therefore, we keep track of the number of sensor visits (sensor covers) of each map cell. The definition of a frontier point is now expanded: a frontier point is a point which represent an unknown region, has at least one open-space neighbor and has not been scanned before. 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 (lines 15–16).

Figure 5 demonstrates the necessity of the above. Figure 5(a) shows frontiers extraction without tracking the number of visits. It can be seen that there are frontiers that lie inside an open space

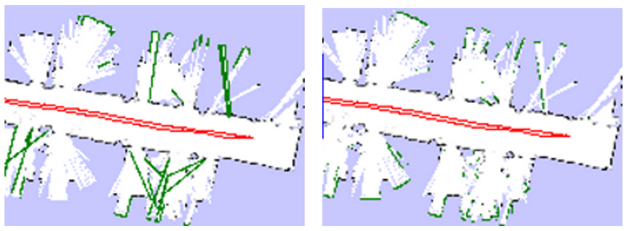

(a) Incorrectly re-detected frontiers. (b) Correct detection.

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

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 avoidance of redetecting same frontiers. It can be seen that every frontier separates known and unknown regions.

4.4.3 *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 24–31 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.

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

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

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

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 was belonged to a frontier, the current scanned point is removed from the frontier and the frontier is spit into two partial frontiers using the current scanned point as a pivot (lines 28–29).

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.

4.4.4 *Storing New Detected Frontiers*

In the last phase of the maintenance step (lines 32– 38) 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.

5. FFD IS SOUND AND COMPLETE

We show that Algorithm 2 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 5.1. *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 a 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 Lines 8–23. 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 8–23 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 if 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 5.2. *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 5.1.

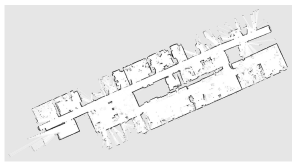
Case 2. f was a new frontier point at time s, where $s < t$. Let s be the earliest time in which f was a frontier. Based on lemma 5.1, it follows that it was detected at this time. All that remains to show is that given the f is still valid at time t, *FFD* will maintain knowledge of it from time s and 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, will 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 lines 24–31. In these, *FFD* scans all points that are covered by the robot's sensors and checks if any points should be eliminated (line 25). 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 5.2 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.


(a) Cartesium Building, University of Bremen.

(b) Freiburg, Building 079.

Figure 6: Some of the testing environments.

THEOREM 5.3. *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 (Lines 8–23) considers only points on the contour, it follows that $\hat{f}$ must be located on the contour *and* detected by lines 8–23. However, line 13 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 line 25 FFD scans all points in the active area and checks if they contain a frontier index. When FFD scans $\hat{f}$, it will find out that it contains a valid frontier index (because it has previously belonged to a valid frontier) and continues executing lines 27–31. In these lines, FFD checks and removes from the DB 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 5.3 holds for any arbitrary point, it follows that FFD never incorrectly marks a non-frontier point as a frontier. It is thus sound.

6. EXPERIMENTAL RESULTS

We have fully implemented *WFD* and *FFD* and performed testings on data obtained from the Robotics Data Set Repository (Radish) [8]. We used *WFD* without the suggested speed-up feature, in order to compare all algorithms fairly. Figure 6 shows a few of the environments used for the evaluation. *WFD* and *FFD* were compared with a *SOTA* (state-of-the-art) frontier detection algorithm, due to Wurm and Burgard [15, 16].

To evaluate the algorithms, we integrated them into a single-robot exploration system. The system is based on GMapping, an open-source SLAM implementation [7]. 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 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 and Random Access Memory (RAM) in size of 4 GB.
- Second experiment: we used a slower desktop computer containing Intel Pentium III (Coppermine) with clock speed of 800 MHz and Random Access Memory (RAM) in size of 1 GB. Research-grade robots typically have a faster CPU, but commercial robots typically do not.

We used several environments taken from Radish [8]:

(A) Cartesium Building, University of Bremen

(B) Freiburg, Building 079

(C) Outdoor dataset recorded at the University of Freiburg, (C)

(D) 3rd Floor of MIT CSAIL

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

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.

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:

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

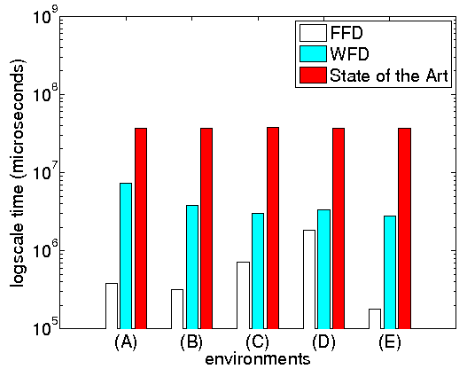
Moreover, we remind the reader that because *FFD* is called for every particle in the particle-filtering GMapping [7], the results here accumulate also over the number of particles (30 in our case).

We begin by examining overall performance. Figure 7 shows one set of 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.

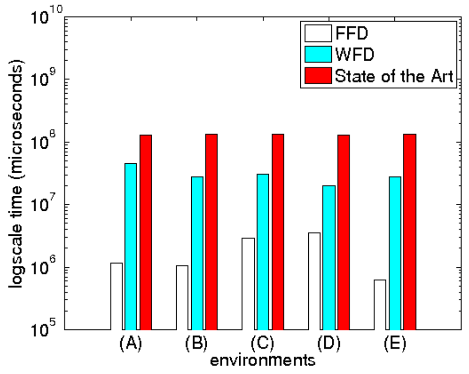
Figure 7 shows that *WFD* is faster than *SOTA* by approximately one order of magnitude. *FFD* is faster than *WFD* by one to two orders of magnitude. Indeed, *FFD* performs close to the one-second line. In contrast, *WFD* and *SOTA* typically take anywhere from 10 to 100 seconds to perform their task, even on relatively fast machines.

FFD's improvement over the others 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 thus turn to evaluating *FFD* at a finer resolution. Figure 8 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.



(a) Intel T6600



(b) Intel Coppermine

Figure 7: Comparing WFD and FFD to State-of-the-Art algorithm on different machines.

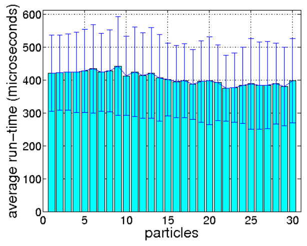
The figure shows that the per-particle run-time is measured in a few hundred micro-seconds. Thus the overall results were accumulating comparing the accumulation of thousands of *FFD* runs against single *WFD* and *SOTA* runs. Indeed, one can boost *FFD*'s execution time by not executing it on every 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.

7. CONCLUSIONS AND FUTURE WORK

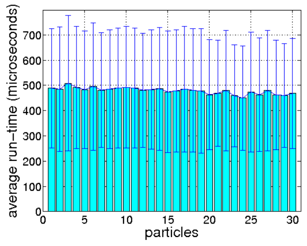
Frontier-based exploration is the most common approach to solve the exploration problem. State-of-the-art frontier detection methods process the entire map data, which hangs the exploration system for a few seconds with every call to the detection algorithm.

We introduced two novel faster frontier detectors, *WFD* and *FFD*. The first, 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, a laser-based approach for frontier detection, only processes new laser readings which are received in real time eliminating also much of the known area search. However, maintaining previous frontiers knowledge requires tight integration with the mapping component, which may not be straightforward. We describe efficient implementation for both algorithms, and compare them empirically. *FFD* is shown to outperform *WFD* and the state-of-the-art by 1–2 (2–3, resp.) orders of magnitude.

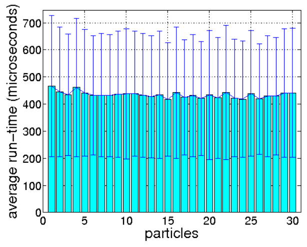
In future, we plan to integrate *FFD* with EKF-based SLAM mappers, which we hope will lead to further improvements. We also



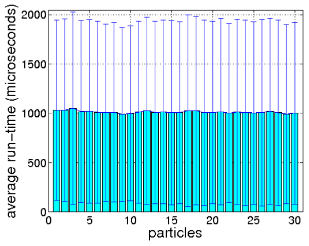
(a) Cartesium Building, Bremen



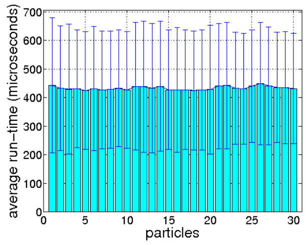
(b) Freiburg, Building 079



(c) Outdoor dataset, University of Freiburg



(d) 3rd Floor of MIT CSAIL



(e) Edmonton Convention Centre

Figure 8: *FFD* run-time for individual SLAM particles.

plan to begin investigation of novel exploration policies, based on real-time frontier-detection.

Acknowledgements. We thank Kai M. Wurm and Wolfram Burgard for providing us with their own implementation of state-of-the-art frontier detection algorithm. Thanks go to Cyrill Stachniss, Giorgio Grisetti and Nick Roy for providing data to the Robotics Data Set Repository (Radish) [8].

8. REFERENCES

[1] M. Berhault, H. Huang, P. Keskinocak, S. Koenig, W. Elmaghraby, P. Griffin, and A. Kleywegt. Robot exploration with combinatorial auctions. In *IROS-03*, pages 1957–1962, 2003.

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

[3] 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.

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

[5] D. Calisi, A. Farinelli, L. Iocchi, and D. Nardi. Multi-objective exploration and search for autonomous rescue robots: Research articles. *J. Field Robot.*, 24:763–777, August 2007.

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

[7] 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.

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

[9] M. Keidar, E. Sadeh-Or, and G. A. Kaminka. Fast frontier detection for robot exploration. In F. Dechesne, H. Hattori, A. ter Mors, J. M. Such, D. Weyns, and F. Dignum, editors, *Advanced Agent Technology: AAMAS 2011 Workshops. Revised Selected Papers*, volume 7068 of *Lecture Notes in Computer Science (LNCS)*, pages 281–294. 2012.

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

[11] R. Sawhney, K. M. Krishna, and K. Srinathan. On fast exploration in 2D and 3D terrains with multiple robots. In *AAMAS-09*, pages 73–80, 2009.

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

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

[14] A. Visser and B. A. Slamet. Including communication success in the estimation of information gain for multi-robot exploration. In *WiOpt-08*, pages 680–687, 2008.

[15] K. Wurm, C. Stachniss, and W. Burgard. Coordinated multi-robot exploration using a segmentation of the environment. In *IROS-08*, Nice, France, Sept. 2008.

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

[17] B. Yamauchi. Frontier-based exploration using multiple robots. In *Agents-98*, pages 47–53, 1998.