Robot Exploration with Fast Frontier Detection: Theory and Experiments

Matan Keidar

MAVERICK Group
Department of Computer Science
Bar-Ilan University

Thesis Defense

Exploration

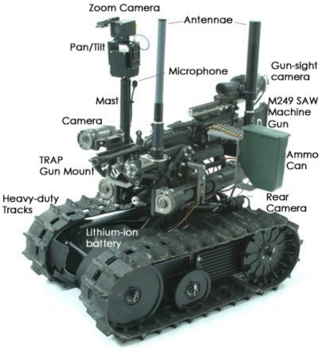
- Robot Exploration: a fundmental problem in mobile robotics
- Variety of applications:
  - Search and Rescue [Kitano et al., 1999]
  - Planetary Exploration [Apostolopoulos et al., 2001]
  - Surveillance [Hougen et al., 2000]

Exploration

- Robot Exploration: a fundmental problem in mobile robotics
- Variety of applications:
  - Search and Rescue [Kitano et al., 1999]
  - Planetary Exploration [Apostolopoulos et al., 2001]
  - Surveillance [Hougen et al., 2000]

Exploration

- Robot Exploration: a fundmental problem in mobile robotics
- Variety of applications:
  - Search and Rescue [Kitano et al., 1999]
  - Planetary Exploration [Apostolopoulos et al., 2001]
  - Surveillance [Hougen et al., 2000]



Frontier-Based Exploration

- The most common approach to exploration is based on *frontiers*
- *Frontier*: separates known regions from unknown regions
  - Set of *unknown* points
  - Each has at least one *open-space* neighbor
- By moving towards frontiers, robots keep discovering new regions
- Yamauchi was the first to show a frontier-based strategy
  - [Yamauchi, 1997, 1998]
- His work preceeded many others
  - [Burgard et al., 2005, Lau, 2003, Sawhney et al., 2009]

Frontier-Based Exploration

- The most common approach to exploration is based on *frontiers*
- *Frontier*: separates known regions from unknown regions
  - Set of *unknown* points
  - Each has at least one *open-space* neighbor
- By moving towards frontiers, robots keep discovering new regions
- Yamauchi was the first to show a frontier-based strategy
  - [Yamauchi, 1997, 1998]
- His work preceeded many others
  - [Burgard et al., 2005, Lau, 2003, Sawhney et al., 2009]

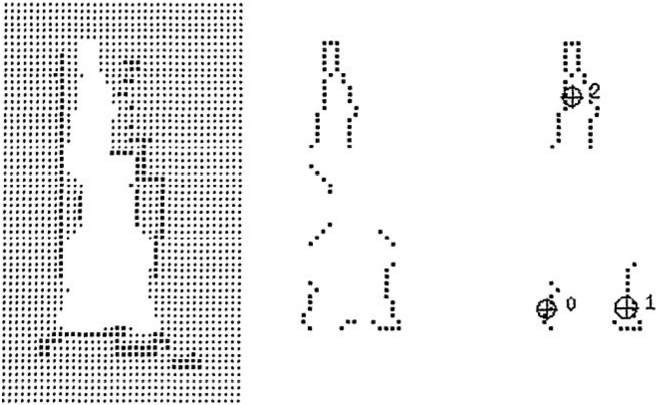


Figure: Image taken from [Yamauchi, 1998]: evidence grid, frontier points, extraction of different frontiers (from left to right).

Outline

- Previous methods for computing frontiers: Slow!
  - searches the entire map with every call
- Two approaches for frontier detection: *WFD*, *FFD*
  - *WFD*: explores only known regions
  - *FFD*: explores only border of known regions
- Theoretical analysis (correctness, complexity)
- Incremental versions of *WFD*: *WFD-INC*, *WFD-IP*
- Results: improvement of 1-2 orders of magnitude over *SOTA*

Existing Frontier Detection is Slow

- Frontier detection algorithms rely on computer vision methods
  - e.g. edge detection and region extraction
- They have to process the entire map data with every execution
- Existing frontier detection methods take ∼10–30 seconds to run
  - Even on powerful computers
  - Exploring a large area forces the robot to wait in its spot
  - Frontier detection is called only when the robot arrives at its target

Result

Efficient frontier detection can shorten the exploration time

Existing Frontier Detection is Slow

- Frontier detection algorithms rely on computer vision methods
  - e.g. edge detection and region extraction
- They have to process the entire map data with every execution
- Existing frontier detection methods take $\sim$10–30 seconds to run
  - Even on powerful computers
  - Exploring a large area forces the robot to wait in its spot
  - Frontier detection is called only when the robot arrives at its target

Result

Efficient frontier detection can shorten the exploration time

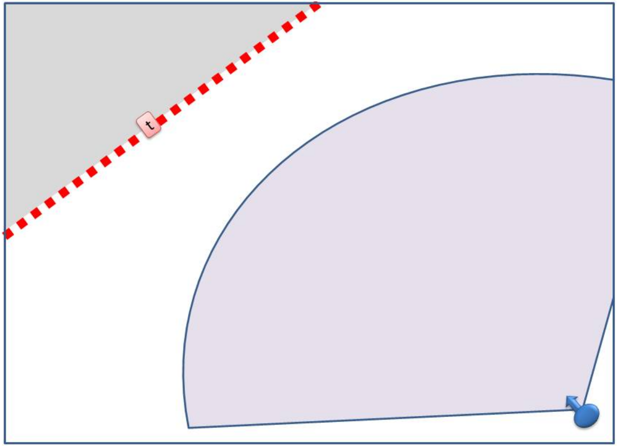
Existing Frontier Detection is Slow

- Frontier detection algorithms rely on computer vision methods
  - e.g. edge detection and region extraction
- They have to process the entire map data with every execution
- Existing frontier detection methods take $\sim$10–30 seconds to run
  - Even on powerful computers
  - Exploring a large area forces the robot to wait in its spot
  - Frontier detection is called only when the robot arrives at its target

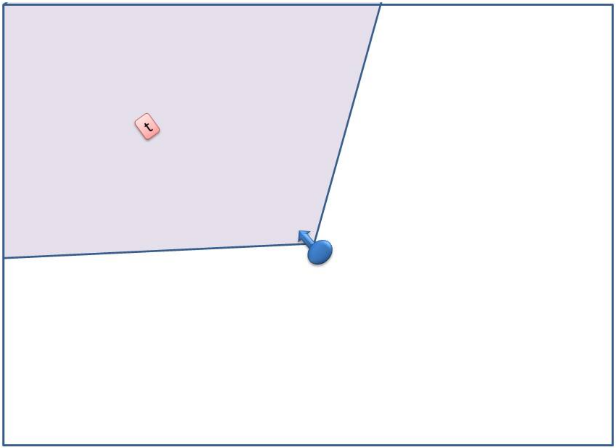
Result

Efficient frontier detection can shorten the exploration time

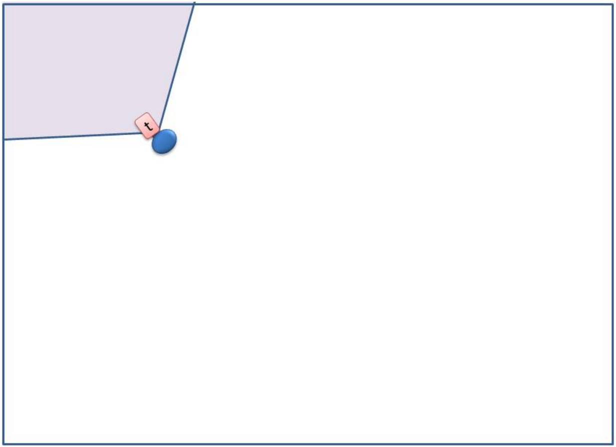
Single-Robot Example



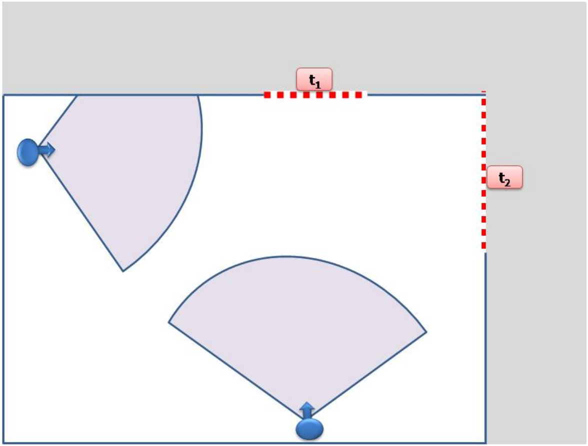
Single-Robot Example



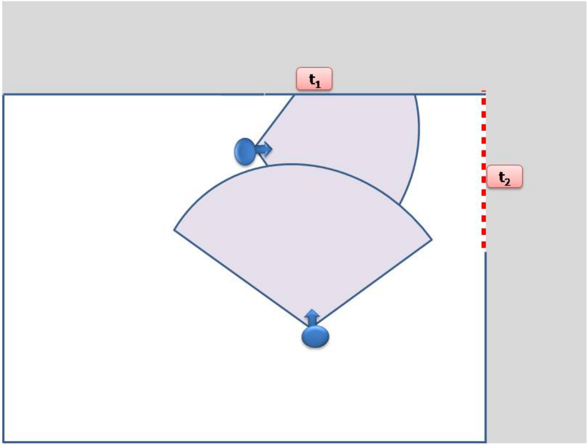
Single-Robot Example



Multi-Robot Example



Multi-Robot Example



Simulaneous Localization and Mapping Methods

- Builds a map of an unknown environment while navigating
- Robots do not utilize any *a priori* knowledge of the environment
- *SLAM* is a concept, not an algorithm
- *Extended Kalman Filter*
  - Beliefs are represented by moments
  - Markov assumptions are hold and the posteriors are Gaussian
  - Only one map that is composed from observed landmarks
- *Particle Filter*
  - Also known as *sequential Monte Carlo method*
  - Online version of *Markov Chain Monte Carlo (MCMC)* methods
  - State: set of variables (e.g map of explored area, robot position)
  - Each particle is an instance of the system state at a certain time

Simulaneous Localization and Mapping Methods

- Builds a map of an unknown environment while navigating
- Robots do not utilize any *a priori* knowledge of the environment
- *SLAM* is a concept, not an algorithm
- *Extended Kalman Filter*
    - Beliefs are represented by moments
    - Markov assumptions are hold and the posteriors are Gaussian
    - Only one map that is composed from observed landmarks
- *Particle Filter*
    - Also known as *sequential Monte Carlo method*
    - Online version of *Markov Chain Monte Carlo (MCMC)* methods
    - State: set of variables (e.g map of explored area, robot position)
    - Each particle is an instance of the system state at a certain time

Simulaneous Localization and Mapping Methods

- Builds a map of an unknown environment while navigating
- Robots do not utilize any *a priori* knowledge of the environment
- *SLAM* is a concept, not an algorithm
- *Extended Kalman Filter*
  - Beliefs are represented by moments
  - Markov assumptions are hold and the posteriors are Gaussian
  - Only one map that is composed from observed landmarks
- *Particle Filter*
  - Also known as *sequential Monte Carlo method*
  - Online version of *Markov Chain Monte Carlo (MCMC)* methods
  - State: set of variables (e.g map of explored area, robot position)
  - Each particle is an instance of the system state at a certain time

WFD: Wavefront Frontier Detector

- Breadth-first search approach for frontier detection
- *WFD* avoids searching unknown regions
- *WFD* scans only known regions

WFD Outline

1. Enqueue current robot position
2. Perform Breadth-First Search
   - scan only open-space points that were not previously scanned
3. For every dequeued point, check if it is a frontier point
   - If True: extract the frontier by using another Breadth-First Search

WFD Conclusions

- Ensures that only known regions are actually scanned
- Frontier points are adjacent to open space points
  - All relevant frontiers will be found when *WFD* finishes
  - Connectivity of frontier points ensures complete frontier extraction
  - The algorithm does not have to scan the entire grid each time
- *WFD* still searches frontiers in all known space

FFD: Fast Frontier Detector

- Insights:
  - New frontiers are never contained within known regions
  - New frontiers are never wholly within unknown regions
- Hence, scanning all known regions is definitely unnecessary
  - and not time-efficient
- *FFD* avoids searching both known and unknown regions
  - Only processes new laser readings

FFD: Fast Frontier Detector

- Insights:
  - New frontiers are never contained within known regions
  - New frontiers are never wholly within unknown regions
- Hence, scanning all known regions is definitely unnecessary
  - and not time-efficient
- *FFD* avoids searching both known and unknown regions
  - Only processes new laser readings

FFD: Fast Frontier Detector

- Insights:
  - New frontiers are never contained within known regions
  - New frontiers are never wholly within unknown regions
- Hence, scanning all known regions is definitely unnecessary
  - and not time-efficient
- *FFD* avoids searching both known and unknown regions
  - Only processes new laser readings

FFD Outline

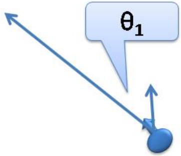
1. Sorting
2. Contour
3. Detecting New Frontiers
4. Maintaining Previously Detected Frontiers

Sorting

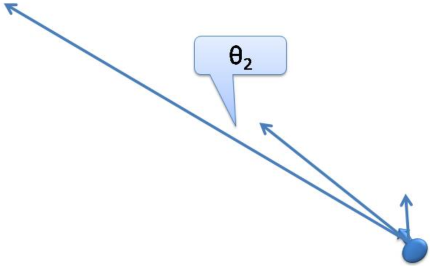
- Most laser sensors return readings that are already sorted
  - Points that are sorted according to polar angle
  - The robot as center
- However, if this is not the case, we can sort them efficiently

Sorting in details

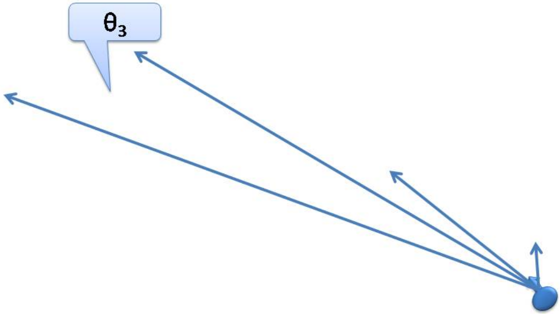
Sorting



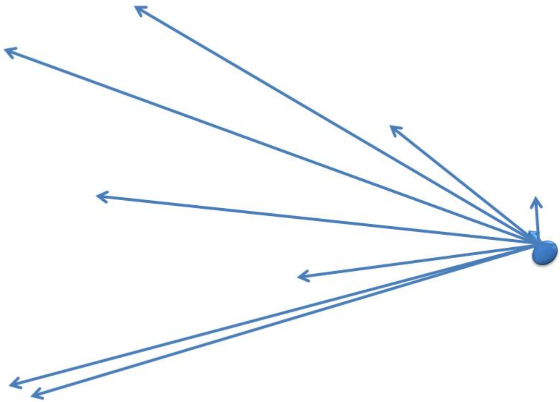
Sorting



Sorting



Sorting



Contour

- **Input**: sorted set of points
- **Output**: a contour that is built from the laser readings set
  - The line that connects each two adjacent points from the set
- Calculate the points that lie between each adjacent laser readings
- The desired contour contains all the points mentioned above

Contour

Problem

We need the algorithm to be fast and robust against rounding errors

Solution

We use *Bresenham's line algorithm* [Bresenham, 1965]

Contour

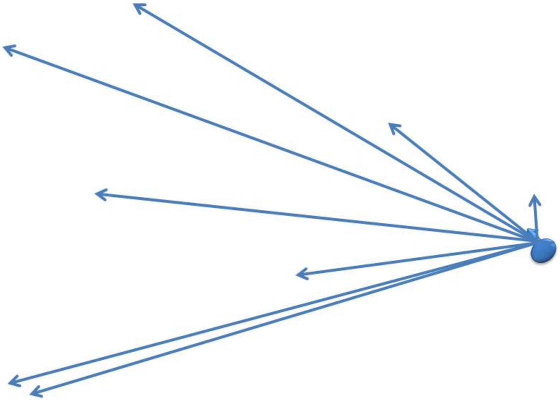
Problem

We need the algorithm to be fast and robust against rounding errors

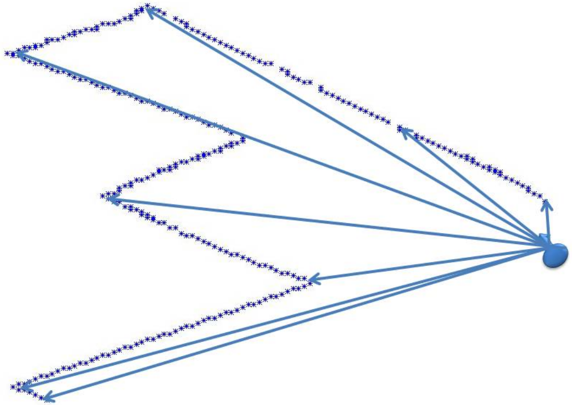
Solution

We use *Bresenham's line algorithm* [Bresenham, 1965]

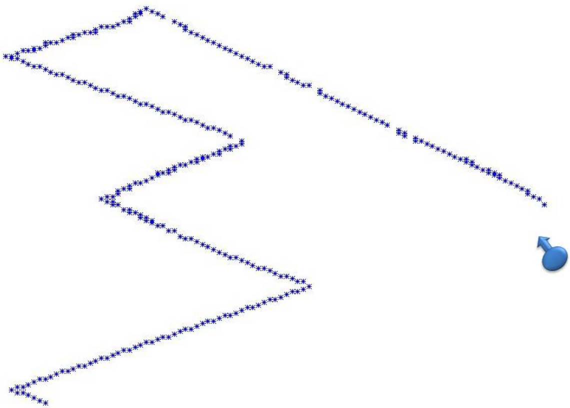
Contour



Contour



Contour



Detecting New Frontiers

- We scan the calculated contour from previous step
- Each point is compared with its (already scanned) predecessor
- Four possible cases:
  - Scanned point is not a frontier cell
  - Scanned point is a frontier cell but its predecessor is not
  - Both scanned point and its predecessor are frontier points
  - Scanned point is not a frontier cell but its predecessor is

Detecting New Frontiers

- We scan the calculated contour from previous step
- Each point is compared with its (already scanned) predecessor
- Four possible cases:
  - Scanned point is not a frontier cell
  - Scanned point is a frontier cell but its predecessor is not
  - Both scanned point and its predecessor are frontier points
  - Scanned point is not a frontier cell but its predecessor is

Maintenance: Motivation

- *FFD* gains its speed by only processing the laser readings
  - rather than entire regions of the map
- Previously detected frontiers are not updated during navigation
- Only the frontier that the robot is headed to

Problem

FFD is able to detect only *new* frontiers in each execution

Solution

Maintenance of frontiers which are not covered in the sensors range

Maintenance: Motivation

- *FFD* gains its speed by only processing the laser readings
  - rather than entire regions of the map
- Previously detected frontiers are not updated during navigation
- Only the frontier that the robot is headed to

Problem

FFD is able to detect only *new* frontiers in each execution

Solution

Maintenance of frontiers which are not covered in the sensors range

Maintenance 1: Eliminating Previously Detected Frontiers

- Points which are no longer in frontiers have to be eliminated
- *Active Area*: blocking rectangle constructed from laser readings
- If a frontier is to be eliminated, it must lie inside the active area
  - it contains regions that are covered by the robot's sensors
- *FFD* scans each point that lies inside the *Active Area*
  - Checking if a point is previously belonged to a frontier is fast

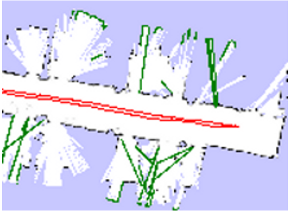
Maintenance 2: Avoiding Re-detection of Same Frontier

Problem

FFD might wrongly detect a new frontier in an already scanned area

Solution

FFD distinguishes between laser readings in different time frames



Maintenance: Summary

- *FFD* has to run in the background
  - In contrast to other approaches that are executed in a certain time
- *FFD* requires robustness against map orientation changes
  - caused by loop-closures

FFD is Complete

Lemma (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.

FFD is Complete

Proof.

- Towards a contradiction: *FFD* did not recognize *f* as a frontier
- *f* cannot be located wholly within an *unknown* region
  - because it must have at least one *Open Space* neighbor
- *f* cannot be located wholly within a *known*
  - since *f* is a valid frontier point and hence, its value is *Unknown*
- Therefore, *f* must be located on the contour itself.
- Detection Stage: *FFD* scans *all* contour points sequentially
  - and specifically searches for frontier points
- It follows that *f* would be detected, contradicting the assumption

□

FFD is Complete

Theorem (Completeness)

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

FFD is Complete

Proof.

- **Case 1.** f **is a new frontier point at time** t**.**
  Trivially, this case is handled directly by lemma 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 1, it follows that it was detected at this time.
    - f must be a frontier point that is maintained by *FFD* at time t.
    - f can be eliminated only by the maintenance stage.
    - f is still a valid frontier and hence, was not covered by the sensors.
    - Therefore, f will not be scanned and eliminated in time t
    - $\Rightarrow$ f remains classified as a frontier by *FFD*.
- In both cases *FFD* will recognize f to be a valid frontier at time t.

□

FFD is Sound

Theorem (Soundness)

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 $\langle O_0, \ldots, O_t \rangle$.

FFD is Sound

Proof.

- $\hat{f}$ is not *Unknown* or all its neighbors are not *Open Space*
- **Case 1. $\hat{f}$ is marked as a new frontier.**
  - *FFD* detects $\hat{f}$ as a new frontier
  - $\hat{f}$ must be located on the contour *and* detected by Extraction stage
  - *FFD* specificially avoids detecting non-frontier points as frontiers
  - Case 1 is not possible
- **Case 2. $\hat{f}$ is an old frontier but was not eliminated.**
  - $\hat{f}$ is located inside the active area and was not eliminated.
  - $\hat{f}$ is a point that was covered by the robot's sensors
  - Each point in the map keeps a frontier index (or NULL value)
  - *FFD* scans $\hat{f}$ and finds out it has a not NULL frontier index
  - The Maintenance stage will eliminate $\hat{f}$
  - Case 2 is not possible

□

WFD Complexity

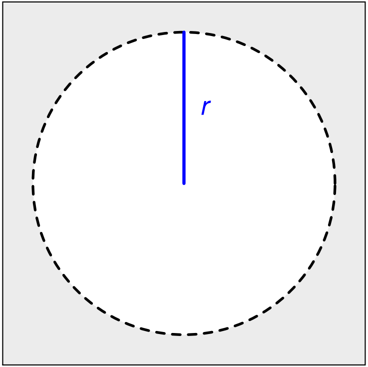
- *WFD* is based on Breadth-First Search (BFS) over the map.
- *WFD* scans all *Open Space* regions for frontier points.
- When a frontier point is found, another BFS is executed
  - in order to extract the frontier.
- BFS time complexity is linear in size of the search space.
- $\Rightarrow$ Linear in size of *area* and *perimeter* of *Open Space* regions

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

WFD Best Case

Best Case

The perimeter of the *open-space* regions is minimal relatively to the area of the *open-space* regions



WFD Best Case

- Notation:
  - $S_{open} :=$ the area of the shape
  - $P_{opt} :=$ the perimeter of the shape
- The best case holds:

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

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

WFD Worst Case

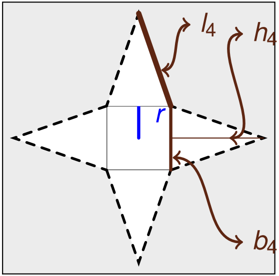
Worst Case

- Maximize the length of the perimeter
  - while keeping the total area of the *open-space* regions.
- Use a polygon as an approximation to a circle.
- The level of accuracy is determined by k, the number of vertices.

WFD Worst Case

Notation (for a given k vertices):

- $S_k :=$ the area of the shape.
  - $S_k = S_{open}$ (area remains the same)
- $P_k :=$ the perimeter of the shape.
- $b_k :=$ the base of the inner *open-space* polygon.
- $h_k :=$ the height of an outer triangle
- $l_k :=$ the length of an outer triangle side.



WFD Worst Case

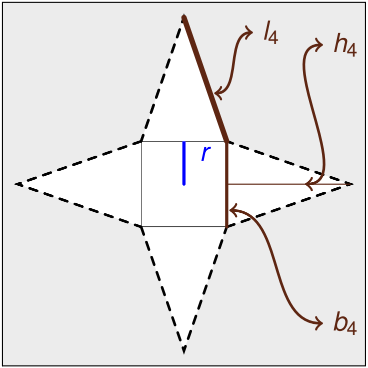


Figure: $k = 4$

WFD Worst Case

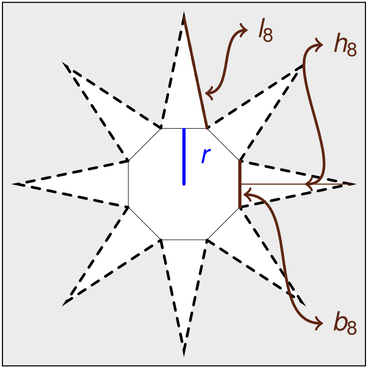


Figure: $k = 8$

WFD Worst Case

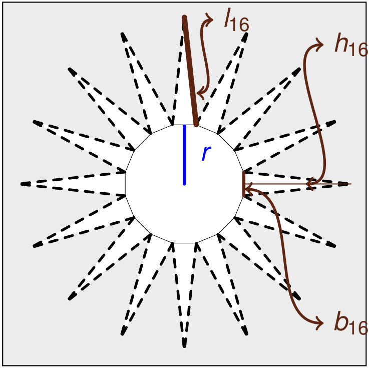


Figure: $k = 16$

WFD Worst Case

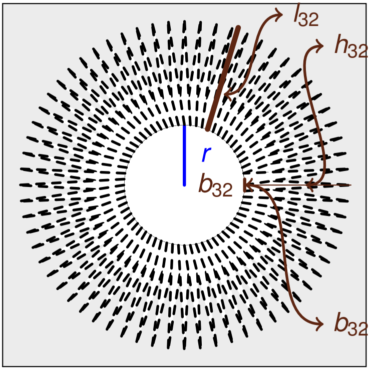


Figure: $k = 32$

WFD Worst Case

- The *open-space* regions are equal:

$$\begin{aligned} S_k &= S_{open} \\ \underbrace{k \cdot \frac{r \cdot b_k}{2}}_{\text{inner polygon}} + \underbrace{k \cdot \frac{h_k \cdot b_k}{2}}_{\text{outer triangles}} &= 4\pi r^2 \end{aligned}$$

- We can express b_k by:

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

- $\Rightarrow h_k$ can be expressed by:

$$h_k = \frac{8\pi r^2}{k \cdot b_k} - r$$

WFD Worst Case

- Each outside triangle edge can be expressed by:

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

- The length of the polygon perimeter equals to:

$$P_k = 2k \cdot l_k$$

- We would like to express P_k as a function of S_{open}:

$$P_k = \sqrt{\left(\frac{2 \cdot S_{open}}{r \cdot \tan\frac{\pi}{k}}\right)^2 - 8\frac{k \cdot S_{open}}{\tan\frac{\pi}{k}} + 4\frac{k^2 \cdot r^2}{\cos^2\frac{\pi}{k}}}$$

WFD Worst Case

- Since $k \leq S_{open}$, P_k can be bounded by:

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

- $\Rightarrow$ Run-time complexity of *WFD* in terms of *open-space* area:

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

FFD Complexity

- It may seem that *FFD*'s complexity is contained within *WFD*
  - since *FFD* searches only inside the active area (*open-space*)
- It is not true since *FFD* has to persistently run in the background
- We analyse the complexity of each stage separatly

FFD Complexity - Sorting Stage

- *FFD* performs polar-sorting of laser readings.
- Cross-product instead of actually calculating angle and radius
  - which are relatively very time-consuming calculations
- Cross-product calculation is performed in a constant time
- Sorting is performed in time of $\mathcal{O}(n \log n)$.
- $l_r :=$ the number of laser readings received in each measurement
- Total complexity of this stage is:

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

FFD Complexity - Contour Stage

- Scans each two adjacent points from the polar-sorted readings.
- *FFD* connects each two adjacent points by a line
  - by calling *Bresenham's line algorithm*, linear time complexity
- d_{p_i,p_j} := euclidean distance between points p_i and p_j
- $c^{(t)}$:= the length of the contour in time t
- Total complexity of this stage is:

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

FFD Complexity - Detection Stage

- Scans the contour extracts new frontiers (if available any)
- Each found new detected frontier is added into a list
- All actions are performed in a constant time
- Total complexity of this stage is:

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

FFD Complexity - Maintenance Stage (1)

Elimination of previous frontiers:

- Scans each point that lies inside the active area
- Checks if it was previously belonged to a frontier
- $A^{(t)} :=$ the bounding rectangle (the active area) in time t
- $f_{max}^{(t)} :=$ length of the longest frontier the frontier database in time t
- $n_f^{(t)} :=$ number of frontiers in the frontier database in time t
- Total complexity of this stage is:

$$\mathcal{O}\left(\underbrace{A^{(t)}}_{\text{scanning active area}} \cdot \left(\underbrace{f_{max}^{(t)}}_{\text{find frontier point}} + \underbrace{\log n_f^{(t)}}_{\text{get frontier from DB}}\right)\right)$$

FFD Complexity - Maintenance Stage (2)

Adding new frontiers:

- Scans each frontier point within all new detected frontiers
- $n_{new}^{(t)} :=$ number of new frontiers that were found in time t
- Checks if new frontier point belongs to previously detected frontier
- The check is performed in a constant time
- All other actions are performed in a constant time
- Total complexity of this stage is:

$$\mathcal{O}\left(\underbrace{n_{new}^{(t)}}_{\text{scan frontiers}} \cdot \left(\underbrace{f_{max}^{(t)}}_{\text{scan all points in frontier}} + \underbrace{\log n_f^{(t)}}_{\text{get frontier from DB}}\right)\right)$$

FFD Complexity - Combining All Stages

- We join all previous stages and get:

$$\mathcal{O}\left(l_r \log l_r + c^{(t)} + \left(A^{(t)} + n_{new}^{(t)}\right) \cdot \left(f_{max}^{(t)} + \log n_f^{(t)}\right)\right)$$

- The above equation can be bounded to:

$$\mathcal{O}\left(c^{(t)} + \log n_f^{(t)}\right)$$

- *FFD* has to run in the background
  - $l_\omega :=$ frequency of the laser sensor
  - $t_m :=$ worst-case elapsed time between two following map-events
  - $P_n :=$ number of particles in the SLAM implementation

$$\mathcal{O}\left(P_n \cdot (t_m \cdot l_\omega) \cdot \left(c^{(t)} + \log n_f^{(t)}\right)\right)$$

General Maintenance Concepts

- *WFD*: detects old and new frontiers within each execution
- *FFD*: is able to detect new frontiers only
  - and therefore has perform a background maintenance
- The maintenance algorithm can be applied to *WFD*
  - and to other frontier detection algorithms

Incremental Wavefront Frontier Detector

- Abbreviation: *WFD-INC*
- Search domain contains only points that lie inside the active area
  - instead of searching the whole known regions for frontier points
- Key idea: the only region that contains changes is the active area
- If a map orientation was changed, then *WFD-INC* acts as *WFD*
- *WFD-INC* has to call the maintenance routine
  - since it does not search the whole map

Incremental-Parallel Wavefront Frontier Detector

- Abbreviation: *WFD-IP*
- Robust against map orientation changes that happens too often
- Key idea: keep a separate instance of *WFD-INC* for each particle
- Execute *WFD-INC* according to each particle's map data
- Result: all *WFD-INC* instances keep frontier data between calls

Experiment Design

- We have fully implemented *WFD*, *WFD-INC*, *WFD-IP* and *FFD*
- We compare all algorithms with a State-of-the-Art algorithm[1]
- Our system is based on *GMapping* SLAM implementation
  - [Grisetti et al., 2005, 2007]
- Two machines:
  - Intel Core 2 Duo T6600 CPU, 2.20GHz, 4GB RAM
  - Intel Pentium III Coppermine CPU, 800MHz, 1GB RAM
- We measured CPU-process time

[1] We thank Kai M. Wurm and Wolfram Burgard for providing us their own implementation

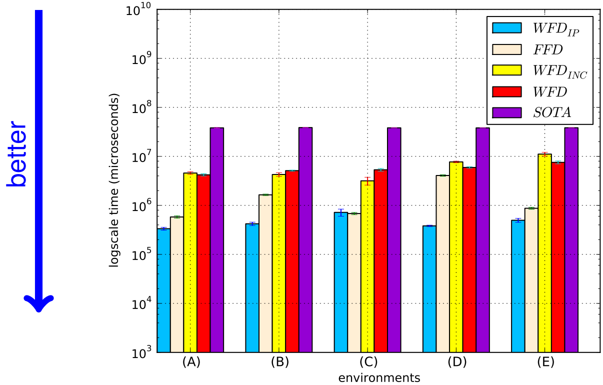
Experiment Design

- All detection algorithms are executed when a map update occurs
- *FFD* is executed when a new laser reading is received
- We accumulate *FFD*'s time between calls to other algorithms
- Tested on data obtained from RADISH [Howard and Roy, 2003]

Experiment Design

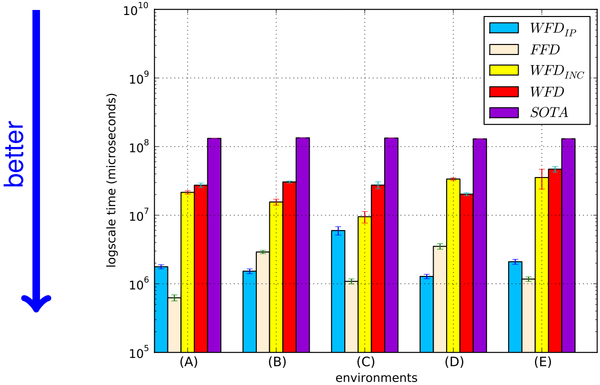
Figure: Example of a testing environment, University of Freiburg. Image was taken from RADISH, Howard and Roy [2003]

Runtime Comparision (Core 2 Duo)



- Y axis: mean execution time (microseconds) on a *logaritmic scale*
- *WFD*, *WFD-INC*: faster than *SOTA* by an order of magnitude
- *FFD*, *WFD-IP*: faster than *SOTA* by two orders of magnitude

Runtime Comparision (Coppermine)



- Y axis: mean execution time (microseconds) on a *logaritmic scale*
- *WFD*, *WFD-INC*: faster than *SOTA* by an order of magnitude
- *FFD*, *WFD-IP*: faster than *SOTA* by two orders of magnitude

What Happend in Environment (D)?

What Happend?

The mean run-time of *WFD* is not relatively slower than *FFD*

Answer

- Environment (D) contains regions that contains small obstacles
- The obstacles enlarge the length of contour scanned by *FFD*
- As shown, the contour length affects the run-time complexity

What Happend in Environment (D)?

What Happend?

The mean run-time of *WFD* is not relatively slower than *FFD*

Answer

- Environment (D) contains regions that contains small obstacles
- The obstacles enlarge the length of contour scanned by *FFD*
- As shown, the contour length affects the run-time complexity

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

What Happend?

- In the worst-case, *WFD-INC* should perform the same as *WFD*
- The run-time means of *WFD* and *WFD-INC* do not have a trend

Hypothesis

- Certain environments have high frequency of particle change
- $\Rightarrow$ *WFD-INC* cleans its previous detected frontiers more often

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

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

What Happend?

- In the worst-case, *WFD-INC* should perform the same as *WFD*
- The run-time means of *WFD* and *WFD-INC* do not have a trend

Hypothesis

- Certain environments have high frequency of particle change
- $\Rightarrow$ *WFD-INC* cleans its previous detected frontiers more often

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

What Happend to *WFD-IP* in Environments (A),(C),(E), Weak Machine?

- Strong machine: *WFD-IP* outperforms all other algorithms
- Weak machine: the situation is different in maps (A),(C),(E)
- Hypothesis (1) Memory:
  - Weak machine is equipped with less RAM than strong machine
  - *WFD-IP* holds a separate instance of *WFD-INC* for each particle
  - Hypothesis: strong machine runs each thread on a separate core
  - In our experiment, 30 instances of *WFD-INC*
- Hypothesis (2) CPU:
  - Weak machine is equipped a single CPU
  - GMapping runs two threads (Main thread and SLAM thread)
  - Hypothesis: strong machine runs each thread on a separate core

What Happend to *WFD-IP* in Environments (A),(C),(E), Weak Machine?

- Strong machine: *WFD-IP* outperforms all other algorithms
- Weak machine: the situation is different in maps (A),(C),(E)
- Hypothesis (1) Memory:
  - Weak machine is equipped with less RAM than strong machine
  - *WFD-IP* holds a separate instance of *WFD-INC* for each particle
  - Hypothesis: strong machine runs each thread on a separate core
  - In our experiment, 30 instances of *WFD-INC*
- Hypothesis (2) CPU:
  - Weak machine is equipped a single CPU
  - GMapping runs two threads (Main thread and SLAM thread)
  - Hypothesis: strong machine runs each thread on a separate core

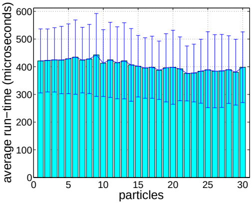
What Happend to *WFD-IP* in Environments (A),(C),(E), Weak Machine?

- Strong machine: *WFD-IP* outperforms all other algorithms
- Weak machine: the situation is different in maps (A),(C),(E)
- Hypothesis (1) Memory:
  - Weak machine is equipped with less RAM than strong machine
  - *WFD-IP* holds a separate instance of *WFD-INC* for each particle
  - Hypothesis: strong machine runs each thread on a separate core
  - In our experiment, 30 instances of *WFD-INC*
- Hypothesis (2) CPU:
  - Weak machine is equipped a single CPU
  - GMapping runs two threads (Main thread and SLAM thread)
  - Hypothesis: strong machine runs each thread on a separate core

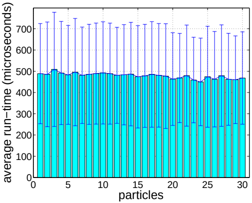
Measuring Run-Time for Individual Particle

- We compare the run-time of individual particles in each map
- Each bar represents a specific particle
- Vertical axis: measures the run-time of *FFD* for the particle
- Error-bars: the standard deviation of each particle's run-time

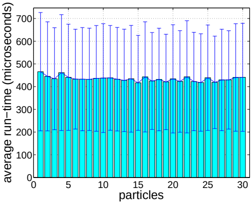
Measuring Run-Time for Individual Particle



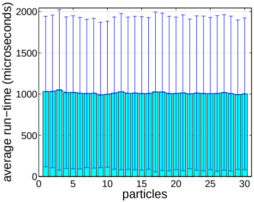
(a)



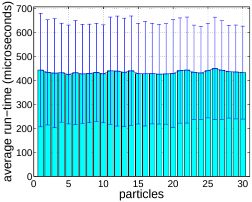
(b)



(c)



(d)



(e)

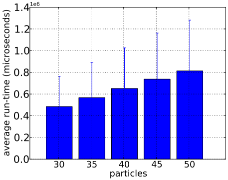
FFD Performance According to Number of Particles

- *FFD* has to persistently run in the background
- Particle-filter systems: each particle has its own instance of *FFD*
- Hence, the overall run-time is increased
- We change the number of particles in different maps
- Each bar represents a run with a specific number of particles
- Vertical axis: mean run-time of *FFD* for the configuration
- Error-bars: standard deviation of each configuration run-time

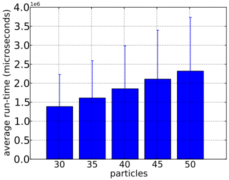
FFD Performance According to Number of Particles

- *FFD* has to persistently run in the background
- Particle-filter systems: each particle has its own instance of *FFD*
- Hence, the overall run-time is increased
- We change the number of particles in different maps
- Each bar represents a run with a specific number of particles
- Vertical axis: mean run-time of *FFD* for the configuration
- Error-bars: standard deviation of each configuration run-time

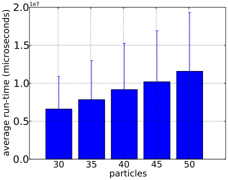
FFD Performance According to Number of Particles



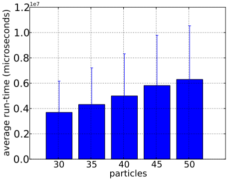
(a)



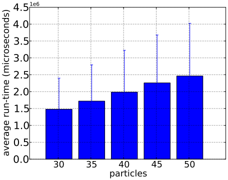
(b)



(c)



(d)



(e)

Summary and Future Work

- Real-Time frontier detection on weak CPU's
- *WFD*, *FFD*, *WFD-INC*, *WFD-IP*: methods for frontier detection
  - significant reduction of calculation time
- Publications:
  - AAMAS 2012
  - ARMS workshop at AAMAS 2011
- Future work:
  - Address efficient methods for maintaining frontiers in *FFD*
  - Novel exploration policies based on real-time frontier-detection

Sorting

- The first step sorts laser readings based on their angle
  - i.e based on the polar coordinates with the robot as the origin
- The naive method for converting Cartesian coordinates to polar coordintates is time-consuming
  - calling *atan2* and *sqrt*
- By using *Cross-Product* we can perform sorting much faster
  - $(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0) \cdot (y_2 - y_0) - (x_2 - x_0) \cdot (y_1 - y_0)$
  - If the result is positive, then $\overrightarrow{P_0P_1}$ is clockwise from $\overrightarrow{P_0P_2}$.
  - Else, it is counter-clockwise.
  - If result is 0, then the two vectors lie on the same line in the plane

Back to *FFD*

Sorting

- The first step sorts laser readings based on their angle
  - i.e based on the polar coordinates with the robot as the origin
- The naive method for converting Cartesian coordinates to polar coordintates is time-consuming
  - calling *atan2* and *sqrt*
- By using *Cross-Product* we can perform sorting much faster
  - $(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0) \cdot (y_2 - y_0) - (x_2 - x_0) \cdot (y_1 - y_0)$
  - 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 result is 0, then the two vectors lie on the same line in the plane

Back to *FFD*

Sorting

- The first step sorts laser readings based on their angle
  - i.e based on the polar coordinates with the robot as the origin
- The naive method for converting Cartesian coordinates to polar coordintates is time-consuming
  - calling *atan2* and *sqrt*
- By using *Cross-Product* we can perform sorting much faster
  - $(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0) \cdot (y_2 - y_0) - (x_2 - x_0) \cdot (y_1 - y_0)$
  - If the result is positive, then $\overrightarrow{P_0P_1}$ is clockwise from $\overrightarrow{P_0P_2}$.
  - Else, it is counter-clockwise.
  - If result is 0, then the two vectors lie on the same line in the plane

Back to *FFD*

Sorting

- The first step sorts laser readings based on their angle
  - i.e based on the polar coordinates with the robot as the origin
- The naive method for converting Cartesian coordinates to polar coordintates is time-consuming
  - calling *atan2* and *sqrt*
- By using *Cross-Product* we can perform sorting much faster
  - $(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0) \cdot (y_2 - y_0) - (x_2 - x_0) \cdot (y_1 - y_0)$
  - 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 result is 0, then the two vectors lie on the same line in the plane

Back to *FFD*

D.S. Apostolopoulos, L. Pedersen, B.N. Shamah, K. Shillcutt, M.D. Wagner, and W.L. Whittaker. Robotic antarctic meteorite search: Outcomes. In *IEEE International Conference on Robotics and Automation*, pages 4174–4179, 2001.

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

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 the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2443–2448, 2005.

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.

Dean F. Hougen, Saifallah Benjaafar, Jordan Bonney, John Budenske, Mark Dvorak, Maria L. Gini, Howard French, Donald G. Krantz, Perry Y. Li, Fred Malver, Bradley J. Nelson, Nikolaos Papanikolopoulos, Paul E. Rybski, Sascha Stoeter, Richard M. Voyles, and Kemal Berk Yesin. A miniature robotic system for reconnaissance and surveillance. In *ICRA*, pages 501–507, 2000.

Andrew Howard and Nicholas Roy. The robotics data set repository (RADISH), 2003. URL `http://radish.sourceforge.net/`.

Hiroaki Kitano, Satoshi Tadokoro, Itsuki Noda, Hitoshi Matsubara, Tomoichi Takahashi, Atsuhi Shinjou, and Susumu Shimada. Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 739–746. IEEE Computer Society, 1999.

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

R. Sawhney, K. M Krishna, and K. Srinathan. On fast exploration in 2D and 3D terrains with multiple robots. In *Proceedings of the 8th*

International Conference on Autonomous Agents and Multiagent Systems. Vol. 1, pages 73–80, 2009.

B. Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, pages 146–151, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8138-1.

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