Robotics and Autonomous Systems 72 (2015) 189–206

Contents lists available at ScienceDirect

Robotics and Autonomous Systems

journal homepage: www.elsevier.com/locate/robot

Incremental algorithms for Safe and Reachable Frontier Detection for robot exploration

P.G.C.N. Senarathne *, Danwei Wang

EXQUISITUS, Centre for E-City, School of Electrical & Electronic Engineering, Nanyang Technological University, Singapore 639798, Singapore

HIGHLIGHTS

- Incremental algorithms for efficient frontier detection in 2D occupancy grid maps.
- Maintains reachability and mapped free space boundary information separately.
- Reachability is incrementally maintained through a safe-patch graph.
- Boundary cells are efficiently maintained as contours in a modified MX-Quadtree.
- Reachable contours are reported as valid frontiers improving detection accuracy.

ARTICLE INFO

Article history:
Received 3 July 2014
Received in revised form
11 April 2015
Accepted 18 May 2015
Available online 18 June 2015

Keywords:
Autonomous exploration
Frontier detection
Reachability detection
Incremental algorithms
Spanning trees

ABSTRACT

Majority of the autonomous robot exploration strategies operate by iteratively extracting the boundary between the mapped open space and unexplored space, *frontiers*, and sending the robot towards the "best" frontier. Traditional approaches process the entire map to retrieve the frontier information at each decision step. This operation however is not scalable to large map sizes and high decision frequencies. In this article, a computationally efficient incremental approach, Safe and Reachable Frontier Detection (SRFD), that processes locally updated map data to generate only the safe and reachable (i.e. valid) frontier information is introduced. This is achieved by solving the two sub-problems of a) incrementally updating a database of boundary contours between mapped-free and unknown cells that are safe for robot and b) incrementally identifying the reachability of the contours in the database. Only the reachable boundary contours are extracted as frontiers. Experimental evaluation on real world data sets validate that the proposed incremental update strategy provides a significant improvement in execution time while maintaining the global accuracy of frontier generation. The low computational footprint of proposed frontier generation approach provides the opportunity for exploration strategies to process frontier information at much higher frequencies which could be used to generate more efficient exploration strategies.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

Autonomous exploration of an unknown environment is a major research problem in robotics. Robotic missions such as autonomous search and rescue [1], planetary mapping [2], underwater mapping [3] require the robot to autonomously map the surrounding unknown environment. During exploration, the robot must sense the environment at a sequence of points until the entire environment is mapped. As planning for the optimal sequence of sensing locations is known to be NP-hard [4,5], all the exploration strategies are reduced to iteratively finding and moving to the next best sensing location. Heuristically, it can be understood that it is best to move the robot towards the boundary between the mapped free space and the unmapped space to expand the map where maps can be represented metrically using grids [6,7] or poly-lines [8,9]. The exploration strategy that generates and evaluates candidate target points based on this heuristic on metric grids is called Frontier Based Exploration [7] and the boundaries identified are called the *frontiers*. This strategy has become the baseline approach for developing exploration missions due to its simplicity and scalability in multi-robot systems.

Frontier based exploration strategy and its many variants have relied on Occupancy Grid Maps [6] to extract the frontier information and have focused mainly on better ways to quantify the desirability of frontier points for robot's next sensing task

* Corresponding author. Tel.: +65 67906350.
E-mail address: namalsenarathne@pmail.ntu.edu.sg (P.G.C.N. Senarathne).

http://dx.doi.org/10.1016/j.robot.2015.05.009
0921-8890/© 2015 Elsevier B.V. All rights reserved.

[4,10,11]. Efficient generation and management of frontier information has largely been ignored in the research literature due to the way the frontier information is traditionally being used by exploration strategies. In these strategies, the selection of a frontier point as the next sensing target location is done only when the robot reaches and finishes sensing at currently selected location. Therefore it does not require frequent generation of frontiers, hence efficient generation of frontiers has not been a major consideration.

While frontiers are extracted sporadically, almost all exploration strategies employ continuous sensing/mapping with map updates occurring at a certain frequency or based on the change of motion (e.g. at 1 Hz or update every 0.5 m motion or 0.1 rad rotation) [12,13]. This continuous mapping results in a frequent evolution of frontiers in the occupancy grid map. It is suggested in the literature that making use of these frequently evolving frontier information could lead to more efficient exploration strategies [14–16]. Having access to frequently evolving frontiers allow robots to quickly uncover dead-end situations and traps. It also allows a robot to discover when the target location assigned is fully or substantially explored by another robot due to unintentional crossing of paths during multi-robot missions [15]. These early discoveries of undesirable situations during exploration missions reduce unnecessary motions for robots thus improves the overall efficiency of missions [17]. Therefore frequent extraction of frontier information from the map is vital in developing more efficient exploration strategies. However, the traditional approach of frontier extraction processes the entire occupancy grid map. Processing the entire map at high frequency and regenerating all frontier cells at each step is not scalable with increasing map sizes along missions. While improving the processing capability of robots is a possible solution, it is not applicable for many robots with computing and payload limitations. Therefore development of efficient algorithms for frontier generation is considered in this work.

The work described in this paper proposes to manage and extract frontier information in an efficient and incremental way by processing only the modified sections of the map at each update step. At each update step, the boundary cells between mapped-free and unknown cells that are safe for the robot are updated using a global database of boundary contours. The reachability of the mapped space is also incrementally updated and is then used to retrieve only the safe and reachable boundary contours as frontiers. This approach alleviates the need to process the entire mapped grid space to compute the reachability of frontiers to filter invalid frontiers (i.e. phantom-frontiers) that are generated due to mapping through small openings or due to errors in mapping as depicted in Fig. 1. Thus it provides a complete incremental approach to generating valid frontier cells.

The rest of the article is organized as follows. In Section 2, the traditional frontier extraction approach is summarized and a review of related incremental algorithms is provided. Section 3 provides the basic definitions used throughout the article. Section 4 details the incremental extraction of safe boundary information between mapped-free and unknown grid-cells. Section 5 formalizes the operations to incrementally maintain the reachability of the mapped area by the robot. Section 6.1 details the implementation of the reachability maintenance operations provided in Section 5 and Section 6.2 details the database used to manage the boundary contour information. Section 7 provides details of the experiments and analyzes the results and Section 8 concludes the article.

2. Literature review

2.1. Traditional frontier cell generation

Most of the exploration strategies require frontier information sporadically with long time intervals in between. Therefore these strategies employ trivial processing of the entire map to generate the frontiers. Initial works [7] describe the generation of frontier cells to be analogous to edge detection and region extraction in computer vision. These steps are clearly described and illustrated in Fig. 3 of [18]. Initially the safe regions in the map are extracted. This involves filtering out grid cells classified as *free* that lead to collisions. Various collision checking algorithms can be employed to extract the safe regions. Approximating the robot's footprint to a circle and dilating obstacles using the radius of this circle [19] can be done to identify the unsafe regions that lead to collisions. The free cells that do not intersect the dilated/inflated obstacle cells can be then retrieved as safe. This is a simple and common approach used for retrieving safe cells for mobile robots operating at slow-medium speeds [20]. Next, only the reachable safe-regions need to be considered to extract frontier information. The reachability of safe-cells could be calculated by either a flood-fill operation [21] with robot's position cells as the seed or by using a distance calculation operation from robot's current position, such as Dijkstra's shortest path algorithm [22] or Lee's algorithm [23]. Then, the boundary between the reachable safe-region and unknown regions are extracted. This could be done by convolving the map with a kernel similar to a one used for edge detection in images. The implementation referred to as the *traditional approach* (TRA), throughout this article, uses obstacle inflation to retrieve safe-regions, Lee's algorithm to retrieve reachability information and finally uses a convolving operation to extract the boundary of reachable safe regions as frontiers.

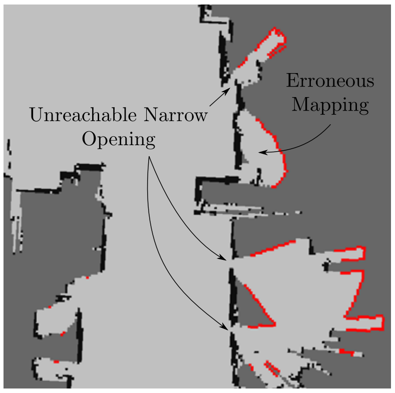


Fig. 1. A section of an occupancy grid map illustrating phantom-frontiers due to erroneous mapping and mapping through narrow openings. (White—free space, Black—Obstacles, Gray—Unknown space, red—phantom frontiers). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

2.2. Different usage of frontier information

Frontier exploration strategies use extracted frontier cell information in different ways. A large collection of exploration strategies use the frontier cell information as it is and evaluate cells individually [24,14]. However, evaluating groups of frontier cells for better exploration performance has also been explored. These include clustering frontier cells based on cluster size [7,25], clustering based on environment segmentation [26,27], representing frontiers cells as contours [28,18] and improving clustering of frontier cells based on semantic information about map features [29,30].

All of the above mentioned clustering/grouping of frontier cells require additional processing after the extraction of initial frontier cell information. Regenerating all frontiers at each update step would thus lead to re-computation of all the cluster information, making frontier extraction process even more computationally inefficient. Recent work on incremental generation and management of dynamic navigational data [31] illustrates the need for development of more efficient algorithms for robotic systems. Therefore, incremental management of all the frontier cells and their cluster/group information becomes even more desirable property for an exploration algorithm.

2.3. Efficient approaches for frontier detection

Some recent works have proposed efficient algorithms for frontier detection. The common approach of all of these frontier detection strategies is to bound the processing of the map to only the updated regions in the map. Two algorithms, Wavefront Frontier Detector (WFD) and Fast Frontier Detector (FFD) are proposed in [15]. WFD is based on the idea of processing only the mapped area instead of the entire map negating the need to unnecessarily process unknown map cells to detect frontiers. A Breadth First Search is initialized from the robot's position cell and is progressed outwards until frontier cells are extracted. This method however degenerates to the traditional approach of processing the entire map as high percentage of the map gets explored by the robot. FFD processes only the current range reading rather than map information. At each sensor range reading, the range values are checked for possible frontier cell information. However, this provides information about only the newly generated frontier cells and not about previously detected frontier information. In order to avoid re-detection of frontiers and elimination of previously detected frontiers which are no longer valid, a separate grid data structure is used to memorize the frontier cell information across sensor range update steps. While FFD is considerably efficient compared to WFD, its non-use of the probabilistic occupancy grid map could restrict its use only to robotic systems equipped with highly accurate range sensors.

WFD-INC [15] is a modification to the WFD method by restricting the breadth first search for frontiers only to the *active region* in the map (i.e. locally updated region). The use of active region is similar to the proposed SRFD approach where only the area bounded by the borders of the newly updated cells are processed. Since WFD-INC is tested on particle based SLAM systems, it requires clearing of previous data and processing of the complete map when the best particle changes. WFD-IP is a simple improvement upon WFD-INC where for each particle a separate instance of WFD-INC is executed alleviating the need to clear old data and restart the detection process. The OFD method proposed in [32] is similar to WFD-INC where only the active region is processed to update frontier information. Instead of using an axially aligned bounding box to bound the active region, oriented bounding boxes [33] are used to further reduce the execution time. However the required back and forth coordinate transforms on a grid result in shifting of detected frontier cells and reduce its accuracy.

FFD and many variants of WFD methods do not specifically mention a step of calculating the safe regions in the environment. In addition, none of these methods contain reachability detection integrated to the frontier extraction process. Therefore, they could output map boundaries that are unreachable by the robot, requiring additional steps to filter these out as evaluation of unreachable frontiers degrade the efficiency of the decision steps of exploration missions. The proposed method includes both incremental management of safe-space and incremental management of reachable space which results in a single integrated set up of detecting only safe and reachable frontiers eliminating the need for separate frontier filtering step. In addition, the extracted frontiers points are reported as contours, further avoiding the need for additional grouping/clustering steps.

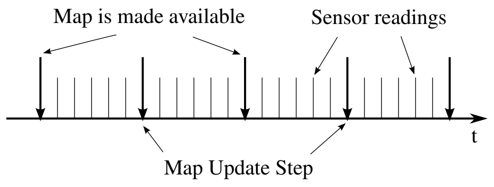


Fig. 2. Relationship between map update step and sensor readings.

3. Preliminaries

This section provides the basic definitions, conventions and assumptions used in the subsequent sections of the article.

3.1. Map update step

Continuous mapping modules used by robotic navigation systems continue to accumulate the received mapping sensor information at a high frequency. A Laser Range Finder (LRF) usually operates at a frequency of around 20–30 Hz. All of these sensor information are used to update the occupancy grid map. The rate at which the map and frontier information should be made available from the mapping module to the rest of the navigation system is highly dependent on the speed of operation of the robot and the nature of the operating environment. Therefore, all of the mapping modules provide a way to alter the frequency of publishing the updated map information for other modules in the navigation system [12,13]. Hence, the map update frequency experienced by the rest of navigation modules is the map publishing frequency. Therefore throughout the rest of this article, *map update step* refers to the phase at which an updated map is made available to the rest of navigation system as depicted in Fig. 2. Table 1 lists the definitions of the major data-structures updated at each such step and the associated abstractions used throughout the article. The subscript k is used to associate an update step with the corresponding symbol. Superscripts *new* and *del* are used to indicate changes to associated sets between two map update steps. For example, $\mathcal{B}_k^{new}$ and $\mathcal{B}_k^{del}$ are used to indicate the cells newly added to $\mathcal{B}_k$ and cells deleted from $\mathcal{B}_{k-1}$ respectively to generate $\mathcal{B}_k$ from $\mathcal{B}_{k-1}$ at update step k.

All of the changes to the map between two map update steps are accumulated in U_k. This is later used as the input to the proposed incremental algorithms to extract frontier information. Since error-free perfect mapping is not assumed, the grid cells are subjected to correctly or incorrectly being marked as obstacles or cleared from being obstacles due to sensor noise and mapping errors. Sets $\mathcal{O}_k^{new}$ and $\mathcal{O}_k^{del}$ are used to track these changes between two map update steps.

3.2. Traversability, reachability and frontiers

Using the abstractions in Table 1 the traversability, reachability and frontiers are defined as follows. A set C, $C \subseteq \mathcal{S}_k$, of cells is *traversable* by the robot if $\forall c_1, c_2 \in C, \exists$ a path in G'_k connecting $\mathcal{V}'(c_1)$ and $\mathcal{V}'(c_2)$. It directly follows that the set of cells represented by each maximal connected sub-graph (i.e. connected component) in G'_k is traversable by the robot. The nodes corresponding to the grid cells of robot's footprint belong to only one of these connected components, denoted by C_k^*. Since the robot is capable of traversing the grid cells associated with the connected component, all the cells associated with C_k^* are considered *reachable* by the robot. While the rest of the grid cells associated with other connected components are traversable, they are not reachable by the robot. This distinction between reachability and traversability is used to define valid frontiers. First, the set of mapped-free-space-boundary cells at step k can be defined as $\mathcal{B}_k = \{c : c \in \mathcal{S}_k \text{ s.t. } \exists\, c' \in \mathcal{N}_4(c) \text{ where } \mathrm{occ}(c') = \mathrm{UNKNOWN}\}$.

Table 1
Notations.

$\mathcal{M}_k$	The set of grid cells of the occupancy grid map after the kth update step
f_m	Map update frequency
$\text{occ}_k(c)$	The occupancy class of grid cell. OBS for obstacle, FREE for free cell and UNKNOWN for unmapped
$us(c)$	The latest update step at which cell c's *occ* value is modified
$\mathcal{O}_k$	The obstacle cell set. $\{c \in \mathcal{M}_k; \text{occ}_k(c) = \text{OBS}\}$
$\mathcal{F}_k$	The set of free cells. $\{c \in \mathcal{M}_k; \text{occ}_k(c) = \text{FREE}\}$
R_r	Radius of the robot
R_{max}	Maximum usable sensor range
$d(c, o)$	Euclidean distance between the grid cells c and o
$\mathcal{I}_k$	The set of inflated obstacle cells, given by $\{c \in \mathcal{M}_k \text{ s.t. } \exists\, o \in \mathcal{O}_k \text{ where } d(c, o) \le R_r\}$
$\mathcal{S}_k$	The set of safe cells for robot, $\{c \in \mathcal{F}_k \setminus \mathcal{I}_k\}$
$\mathcal{N}_x(c)$	x-connected neighboring cells of cell c, $x = \{4, 8\}$
G'_k	Graph representation of the connectivity information of the safe cells $\mathcal{S}_k$; $G'_k = \{V'_k, E'_k\}$
V'_k	Nodes of G'_k, each node represents one cell in $\mathcal{S}_k$
E'_k	Edges of G'_k, based on 8-connectivity
$c(u)$	The grid cell associated with node $u \in V'_k$
$\mathcal{V}'(c)$	Maps safe grid cell c to its associated node in G'_k
$\mathcal{C}_k$	The set of connected components of G'_k
$A'(C)$	The set of grid cells in $\mathcal{S}_k$ associated with the connected component $C \in \mathcal{C}_k$
C^*_k	Connected component reachable by the robot
U_k	The set of cells who's *occ* value is updated during step k
$\mathcal{B}_k$	The set of mapped-free-space boundary cells
$C_{\mathcal{B},k}$	The set of mapped-free-space boundary contours
Γ_k	The set of frontier cells
$\mathcal{X}^i_j$	Nodes added at step j that belong to $C^i \in \mathcal{C}_k$
$\mathcal{Y}^{i,l}_j$	The lth connected component of set $\mathcal{X}^i_j$
$A(\mathcal{Y}^{i,l}_j)$	Set of safe grid cells represented by $\mathcal{Y}^{i,l}_j$
G_k	Graph representation of the connectivity between all $\mathcal{Y}^{i,l}_j$ sets; $G_k = \{V_k, E_k\}$
$\mathcal{T}_k$	Set of spanning trees of G_k
$\mathcal{V}(c)$	Maps grid cell c to a node in V_k
$\mathcal{N}^u$	Neighboring nodes of $u \in V_k$

Then the set of frontier cells present in the map is identified as $\Gamma_k = \{c : c \in \{\mathcal{B}_k \cap A'(C^*_k)\}\}$. Therefore, frontier cells are mapped-free-space boundary cells that are reachable by the robot from its current location.

4. Incremental extraction of mapped free space boundary information

4.1. Incremental identification of safe cells

Incremental extraction of the mapped free space boundary cells requires the identification of safe cells in the occupancy grid map in an incremental way. Considering the definition of safe cell set $\mathcal{S}_k = \mathcal{F}_k \setminus \mathcal{I}_k$, determining the inflated obstacle cells $\mathcal{I}_k$ incrementally is sufficient as the set $\mathcal{F}_k$ denoting free grid cells is already available through the updated map information. Since obstacle grid cells can appear and disappear from the map (Section 3.1), the incremental inflated obstacle detection algorithm must consider both inflation and deflation of obstacle cells. A version of Dynamic Brushfire algorithm [34,31] where wavefront propagation is restricted to R_r is used to process the sets $\mathcal{O}^{new}_k$ and $\mathcal{O}^{del}_k$ to incrementally generate the inflated obstacle cell set from $\mathcal{I}_{k-1}$ to $\mathcal{I}_k$. Overview of the operation is illustrated in Fig. 3.

4.2. Extracting mapped free space boundary information

The updated cell information from U_k is processed to extract information about both disappeared boundary cells and new boundary cells. The safe-area information is used to restrict the extraction of new boundary cells to only safe cells.

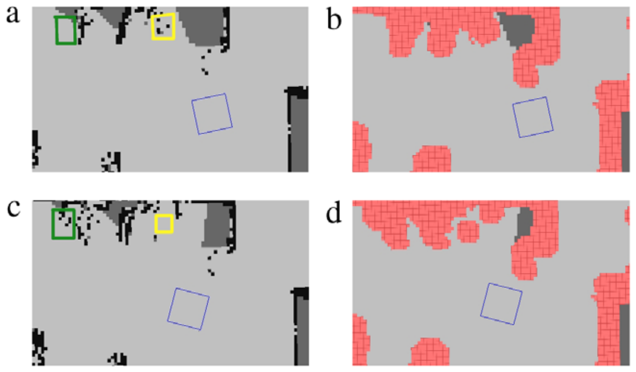


Fig. 3. Incremental management of obstacle inflation. (a) Occupancy grid map and (b) the map with inflated obstacles superimposed at step k. At next update step $k+1$, (c) some new obstacle cells are added (e.g. area marked by green box) and some old obstacle cells are deleted (e.g. area marked by yellow box). (d) The incremental changes to the obstacle inflation are indicated. Blue box is the footprint of the robot. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.2.1. Identifying disappeared mapped space boundary cells

Some mapped-free-space boundary cells extracted in previous update steps disappear during the new map update step. These cells must be identified to be deleted from the database. According to definition of mapped-free-space boundary cells, there are two ways in which a boundary cell could disappear. These two cases are listed below.

1. The cell gets classified as an obstacle cell.
2. All of its 4-connected neighboring cells get mapped as either obstacle or free cells.

It can be seen that all cells that belong to the first case are guaranteed to be in the set U_k of updated cells. However, since the boundary cell is not updated and only its neighboring cells are updated in the second type, the disappeared boundary cell is not guaranteed to be in the set U_k. Therefore, the set U_k is augmented to include all the neighboring cells of existing cells of U_k. This guarantees that all the disappeared boundary cells are in the augmented set U_k when it is processed. The extracted set of disappeared boundary cells are denoted by $\mathcal{B}^{del}_k$ and is deleted from the database.

4.2.2. Extracting new mapped free space boundaries as contours

All the newly updated cells in U_k that satisfy the condition of being a boundary cell are extracted to the set $\mathcal{B}^{new}_k$ as newly generated boundary cells. While these new boundary cells can be treated individually for storage, the connectedness of the boundary cells to each other to create contours can be utilized for efficient extraction and deletion of boundary cells from the database.

According to the definition of boundary cell set $\mathcal{B}_k$, 4-connectivity is used to identify boundary cells in order to retrieve single cell thick boundary contours. Then the identified boundary cells can be chained using a depth-first-search based operations to trace each boundary contour. A typical operation of boundary cell chaining is illustrated in Fig. 4. Any new boundary cell from $\mathcal{B}^{new}_k$ that is not chained to a contour can be used to initialize the chaining process. Then the two neighboring cells are identified to start the chaining of cells in the two opposite directions (Fig. 4(a)). Chaining is continued until no neighboring new boundary cells are detected in both sub chains as depicted in Fig. 4(b). The two sub chains are merged to form a single mapped-free-space boundary contour. The two corner cells of the extracted contour are depicted by the filled circle markings. At the end of contour extraction step, all the extracted boundary contours $C^{new}_{\mathcal{B},k}$ are added to the contour database.

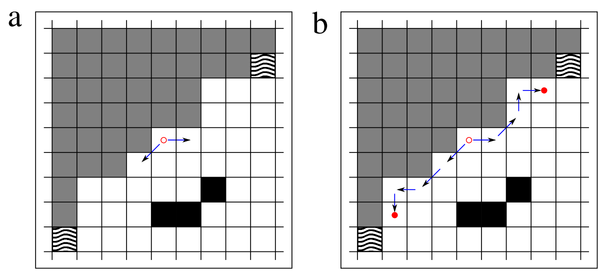


Fig. 4. Boundary contour extraction. (a) Initializing the cell chaining with any boundary cell. (b) Continuing cell chaining in two opposite directions until contour corners are encountered. Cells with wavy lines could be either obstacles or valid boundary cells from previous steps.

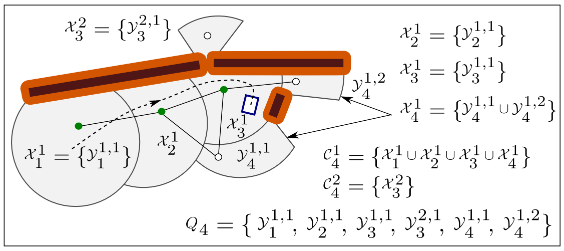

$\mathcal{X}_3^2 = \{\mathcal{Y}_3^{2,1}\}$
$\mathcal{X}_2^1 = \{\mathcal{Y}_2^{1,1}\}$
$\mathcal{Y}_4^{1,2}$
$\mathcal{X}_3^1 = \{\mathcal{Y}_3^{1,1}\}$
$\mathcal{X}_4^1 = \{\mathcal{Y}_4^{1,1} \cup \mathcal{Y}_4^{1,2}\}$
$\mathcal{X}_3^1$
$\mathcal{X}_1^1 = \{\mathcal{Y}_1^{1,1}\}$
$\mathcal{X}_2^1$
$\mathcal{Y}_4^{1,1}$
$\mathcal{C}_4^1 = \{\mathcal{X}_1^1 \cup \mathcal{X}_2^1 \cup \mathcal{X}_3^1 \cup \mathcal{X}_4^1\}$
$\mathcal{C}_4^2 = \{\mathcal{X}_3^2\}$
$Q_4 = \{\mathcal{Y}_1^{1,1}, \mathcal{Y}_2^{1,1}, \mathcal{Y}_3^{1,1}, \mathcal{Y}_3^{2,1}, \mathcal{Y}_4^{1,1}, \mathcal{Y}_4^{1,2}\}$

Fig. 5. Example generation of safe-patches up to update step 4. The associated graph is superimposed (Green and white colored nodes and edges between them, green—visited nodes, white—non-visited nodes). The dotted line indicates the robot's traversed path. The maroon colored blobs and brown colored blobs represent obstacles and the inflated obstacle cells respectively. The shaded area is the mapped free space. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5. Incremental detection of reachability

This section details the development of an incremental algorithm to maintain the reachability information of the map for the robot. This is later used to extract only reachable boundary cells as frontiers according to definition of Γ_k. The formulation is based on generating a graph of traversable "safe-patches" and maintaining their interconnectedness through a collection of dynamic spanning trees to retrieve the reachability information.

At each map update step k, a set U_k of cells are updated. Hence it can be deduced that at any given update step k an existing $A'(C_k^i)$ of connected component $C_k^i \in \mathcal{C}_k$ of G'_k, is generated by a discrete sequence of updates up to step k. Therefore, the associated connected component i at step k, C_k^i, can be partitioned as $P_k^i = \{\mathcal{X}_1^i, \mathcal{X}_2^i, \dots, \mathcal{X}_k^i\}$ according to the update step values of grid cells $us(c)$ with $\forall u \in \mathcal{X}_j^i, us(c(u)) = j$ for $j = 1 \dots k$. For example, $C_4^1 = \{\mathcal{X}_1^1 \cup \mathcal{X}_2^1 \cup \mathcal{X}_3^1 \cup \mathcal{X}_4^1\}$ in Fig. 5.

Given a $\mathcal{X}_j^i \in P_k^i$, for some $C_k^i \in \mathcal{C}_k$, each lth connected component of $\mathcal{X}_j^i$, $\mathcal{Y}_j^{i,l} \subseteq \mathcal{X}_j^i$, $l = 1 \dots |Q_j^i|$, represents a single patch of traversable safe grid cells generated at step j. Therefore, each $\mathcal{X}_j^i$ can be partitioned according to its connected components, i.e. $\mathcal{X}_j^i = \bigcup_{Q_j^i} \mathcal{Y}_j^{i,l}$, where Q_j^i is the set of connected components of $\mathcal{X}_j^i$. E.g. $\mathcal{X}_4^1 = \{\mathcal{Y}_4^{1,1} \cup \mathcal{Y}_4^{1,2}\}$ and $\mathcal{X}_3^1 = \{\mathcal{Y}_3^{1,1}\}$ in Fig. 5. The set of grid cells represented by each connected component $\mathcal{Y}_j^{i,l}$ is denoted by $A(\mathcal{Y}_j^{i,l})$ and is referred to as a "safe-patch". The collection of all $\mathcal{Y}_j^{i,l}$ is denoted by Q_k and is defined as $Q_k = \bigcup Q_j^i; j = 1 \dots k, i = 1 \dots |\mathcal{C}_k|$. Each element in Q_k represents the connected sub-graph in G'_k of the associated safe-patch.

Using the above definitions, a graph $G_k = (V_k, E_k)$ is defined as an abstraction of G'_k using safe-patches. Each node $v \in V_k$ represents a safe-patch $A(\mathcal{Y})$, where $\mathcal{Y} \in Q_k$. The function $Y : V_k \mapsto Q_k$ maps each node in graph G_k to an element in Q_k which represents a safe-patch. An edge $e = (v_1, v_2)$ is defined to be in $E_k \iff \exists$ a $v_3 \in Y(v_1)$ and $v_4 \in Y(v_2)$ s.t. $(v_3, v_4) \in E'_k$. The function $\mathcal{V} : \mathcal{S}_k \mapsto V_k$ provides the mapping between a safe grid cell and its associated node in the graph G_k. Given a node $v \in V_k$, $\mathcal{N}^v$ is defined as the set of neighboring nodes of v. $\mathcal{N}^v = \{v' \in V_k : (v, v') \in E_k\}$. Fig. 5 illustrates an example graph representing the connectivity of safe-patches.

5.1. Checking the reachability of a grid cell

Using the above definitions, the following proposition provides an operation to check the reachability of safe cells in the occupancy grid map using the graph G_k.

Proposition 1. *Given grid cells c_1 and c_2, $\exists$ a traversable path for robot from c_1 to c_2 at step k, if $\mathcal{V}(c_1)$ and $\mathcal{V}(c_2)$ belong to the same connected component in G_k. Therefore, given the robot's position grid cell $r \in \mathcal{S}_k$ and a grid cell $c \in \mathcal{S}_k$, cell c is reachable by the robot if $\mathcal{V}(r)$ and $\mathcal{V}(c)$ belong to the same connected component in G_k.*

Proposition 1 is a simple restatement of connectivity of nodes through paths in graph theory [22].

5.2. Checking for reachability across update steps

During map update steps, new safe patches are generated and existing safe-patches are split to multiple safe patches due to the presence of newly discovered obstacles. Therefore connectivity among nodes representing safe patches evolves with each map update step and must be maintained incrementally in order to extract reachability information. Connectivity queries in graphs are efficiently resolved using spanning trees. The dynamic nature of the graph G_k requires management of dynamic spanning trees. However, the existing approaches do not support full dynamic graph connectivity management and are restricted to a fixed graph size [35] or only to increasing graphs without deletions [36]. Therefore, a new data-structure that aids incremental management of connectivity among nodes in G_k is introduced in the following sections.

5.2.1. Representing the reachable connected component

The reachable connected component in the graph G_k is defined to be the connected component that contains the node that represents the safe patch the robot currently resides. Since the robot's exploratory motion is continuous, this motion can be abstracted to robot visiting a sequence of connected safe patches thus visiting their associated nodes. A node $v \in V_k$ is defined to be a visited node if $A(v) \cap \mathcal{P}_k \neq \phi$, where $\mathcal{P}_k \subset \mathcal{S}_k$ is the set of grid cells representing robot's trajectory so far.

Since visited safe patches are connected, it follows that the visited nodes are connected too and belong to the reachable connected component. This *visited* information can be used to store these nodes in a simple tree where every node has the same root node v. The set of nodes which are not visited by the robot, yet belong to the same reachable connected component can be attached to the visited nodes to generate a spanning tree for the reachable connected component as illustrated in Fig. 6.

The reachability detection of safe-patches represented by non-visited nodes requires multiple search steps to access the root node of the associated spanning tree. Hence, for nodes representing reachable non-visited safe-patches, the maximum shortest-distance to its closest visited-node provides a bound on the reachability detection search steps. Considering v_r to be the constant speed of the robot and f_m to be the map update frequency, the distance the robot travels between two map update steps can be calculated as $\delta d = v_r / f_m$. Considering the common situation of

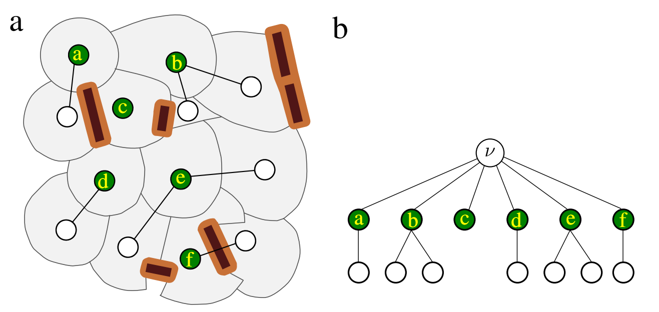


Fig. 6. (a) A set of connected safe-patches and their associated nodes. Green colored nodes represent visited nodes. White colored nodes are not visited yet, but are in the reachable connected component, hence connected to visited nodes. (b) The spanning tree representing the reachable connected component. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

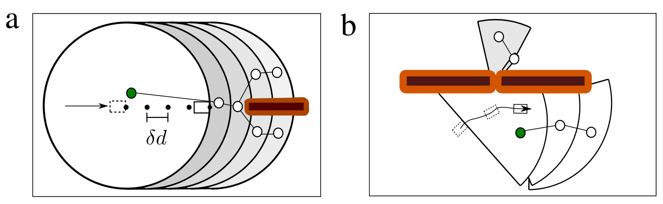


Fig. 7. (a) Illustrating a robot's (rectangle) motion inside a visited safe-patch (white), in the arrow's direction, and generation of non-visited safe-patches (shaded). (b) Bounding the progression of non reachable safe-patches beyond small openings.

$\delta d < R_{max}$, the maximum number of consecutive update steps s_u the map goes through without the robot moving to a different safe-patch is given by $s_u = \lfloor R_{max}/\delta d \rfloor = \lfloor f_m R_{max}/v_r \rfloor$. It directly follows that the upper bound on the shortest-distance from a non-visited reachable node to a visited node in G_k is s_u as illustrated in Fig. 7(a). Hence, the maximum number of search steps to retrieve the root of the reachable connected component ν is bounded to $s_u + 1$.

5.2.2. Representing non-reachable connected components

Non-visited nodes that do not belong to the reachable connected component can be extracted to separate disjoint connected components and are represented using spanning trees. The safe patches represented by such nodes occur when sensor maps areas that are visible through smaller openings, that the robot is unable to travel through. The progression of connected components representing unreachable safe-patches is bounded based on the robot's inability to move beyond the smaller opening in the environment as depicted in Fig. 7(b). Therefore the depth of such a connected component is also bounded by s_u calculated in the previous section. During any update step, if any of these non reachable connected components gets connected to the reachable connected component, their representative spanning trees can be merged to the spanning tree representing the reachable connected component.

5.3. Maintenance of the proposed data structure

Considering that at step $k-1$, the connected components of graph G_{k-1} are represented using a set of spanning trees $\mathcal{T}_{k-1}$, this section formalizes the set of operations that modify the set $\mathcal{T}_{k-1}$ of spanning trees to generate the set $\mathcal{T}_k$ of spanning trees that represent the connected components of graph G_k. The order of invocation of these operations and their dependencies are illustrated in Fig. 8 and are summarized below.

First, the safe-patches that lose full traversability due to new obstacle information are identified and are split to individual safe-patches (Fig. 9) and their corresponding nodes $\hat{\Upsilon}_k$ are generated.

Algorithm 1 UPDATEV(s, u)

```
1: $F \leftarrow \phi$
2: ENQUEUE($F$, $s$)
3: $v \leftarrow$ GENERATE-NEW-NODE() and $\mathcal{N}^v \leftarrow \phi$
4: $\mathcal{V}(s) \leftarrow v$
5: while $\sim$ EMPTY($F$) do
6:     $c \leftarrow$ DEQUEUE($F$)
7:     for all $n \in \mathcal{N}_8(c)$ do
8:         if $n \in \mathcal{S}_k$ AND $\mathcal{V}(n) = u$ then
9:             $\mathcal{V}(n) \leftarrow v$
10:            ENQUEUE($F$, $n$)
11:        else if $\mathcal{V}(n) \neq$ NIL AND $\mathcal{V}(n) \neq v$ then
12:            $\mathcal{N}^v \leftarrow \mathcal{N}^v \cup \{\mathcal{V}(n)\}$
13: return $\{v, \mathcal{N}^v\}$
```

The nodes corresponding to split safe-patches Υ_k^s are removed from $\mathcal{T}_{k-1}$. The nodes in $\hat{\Upsilon}_k$ are inserted to the resultant trees to generate the collection of trees $\dot{\mathcal{T}}_k$. Next, safe patches due to new safe cells $\mathcal{S}_k^{new}$ and their associated nodes Υ_k' are generated and inserted to $\dot{\mathcal{T}}_k$ to generate the collection of trees $\ddot{\mathcal{T}}_k$. The remaining new nodes which are not used for tree augmentation, Υ_k^I, due to these nodes not having any neighboring nodes are represented in isolated spanning trees $\hat{\mathcal{T}}_k$. The collection of trees in $\ddot{\mathcal{T}}_k$ need to be checked for possible connectivity among different trees and any two connected trees are merged. Finally the union of the set of trees generated from the merging operation and $\hat{\mathcal{T}}_k$, results in the collection of spanning trees $\mathcal{T}_k$ that represent the connected components in G_k. The individual update steps are formalized in detail in the following sub-sections.

5.3.1. Updating safe-patch information

Extracting or modifying safe-patch information is central to all the incremental operations in transforming $\mathcal{T}_{k-1}$ to $\mathcal{T}_k$ with new map data and is formalized as follows.

Proposition 2. *Given a seed cell $s \in \mathcal{S}_k$ and its current associated node $\mathcal{V}(s) = u$, executing a flood-fill operation on connected grid cells where the connectivity is restricted to safe cells that share the same current associated node u, extracts a single safe-patch $A(v)$ and its new abstracting node $v \in V_k$.*

The Algorithm 1 details the steps in extracting a safe-patch given a seed cell s and its associated node u. The check in line 8 ensures that only the safe connected cells that share the associated node u are extracted and are associated with the new node v. The returned set $\mathcal{N}^v$ contains all the neighboring nodes of v. Algorithm 1 is used in Lemma 1 (Section 5.3.2) to derive an operation to identify safe-patches $A(v_1), A(v_2), \ldots$ generated after splitting an existing safe-patch $A(u)$ due to changes to inflated obstacle data. Lemma 2 (Section 5.3.4) utilizes Algorithm 1 to derive an operation to extract new safe-patches from new safe cells $\mathcal{S}_k^{new}$. Therefore, $A(v) \neq A(u)$ for all u and v.

5.3.2. Updating safe-patches with new obstacle information to generate $\hat{\Upsilon}_k$

Changes to inflated obstacle information, $\{\mathcal{I}_k^{new} \cup \mathcal{I}_k^{del}\}$, may modify existing safe-patches by shrinking their sizes or by making them non-connected. If a safe-patch becomes non-connected, the safe-patch is considered to be split to multiple new safe-patches. Shrinking a safe-patch can also be considered as splitting it in to one new safe-patch and one non-safe-patch. Hence, both of these operations require the node representing old safe-patch to be removed from the spanning tree and to generate/extract the new safe-patches and adding the corresponding nodes to the spanning tree. Given a node $u \in V_{k-1}$ and its associated safe-patch $A(u)$,

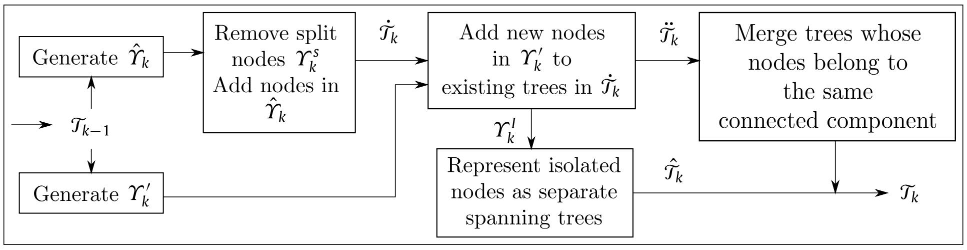

Generate $\hat{\Upsilon}_k$
Remove split nodes Υ^s_k Add nodes in $\hat{\Upsilon}_k$
$\dot{\mathcal{T}}_k$
Add new nodes in Υ'_k to existing trees in $\dot{\mathcal{T}}_k$
$\ddot{\mathcal{T}}_k$
Merge trees whose nodes belong to the same connected component
$\mathcal{T}_{k-1}$
Υ^I_k
Represent isolated nodes as separate spanning trees
$\hat{\mathcal{T}}_k$
Generate Υ'_k
$\mathcal{T}_k$

Fig. 8. The order of update operations conducted to maintain the spanning tree data structure.

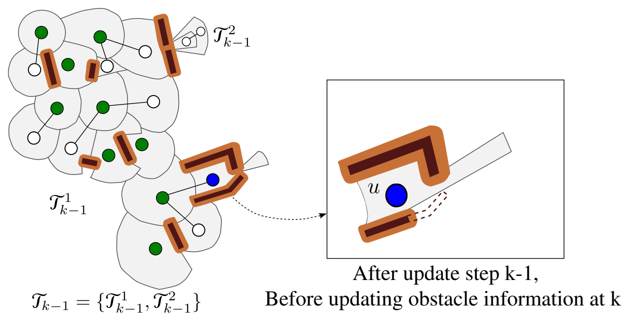

$\mathcal{T}^2_{k-1}$
$\mathcal{T}^1_{k-1}$
u
After update step k-1,
Before updating obstacle information at k
$\mathcal{T}_{k-1} = \{\mathcal{T}^1_{k-1}, \mathcal{T}^2_{k-1}\}$

Fig. 9. Splitting a safe patch due to new obstacle information, depicted in dashed lines. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

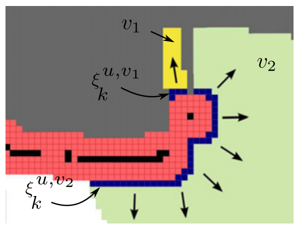


Fig. 10. Using inflated obstacle boundary cells in ξ^{u,v_1}_k and ξ^{u,v_2}_k (blue cells) to split $A(u)$ and generate $A(v_1)$ and $A(v_2)$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the only way that $A(u)$ gets split to multiple disjoint safe-patches during update step k is if some grid cells in $\{\mathcal{I}^{new}_k \cup \mathcal{I}^{del}_k\}$ break the connectedness of $A(u)$. Such a situation is illustrated in Fig. 9 where the safe-patch represented by the dark blue node gets partitioned according to new obstacle information. Hence, it follows that each disjoint safe-patch generated from such a splitting has a set of boundary cells neighboring some cells in $\{\mathcal{I}^{new}_k \cup \mathcal{I}^{del}_k\}$.

The set of boundary cells of inflated obstacle cells generated for step k, $\mathcal{I}^{B,new}_k$, is defined as $\{c : c \in \mathcal{S}_k \text{ s.t. } us(c) = k$ and $\exists\, (\mathcal{V}'(c),\ \mathcal{V}'(i)) \in E'_k$ where $i \in \mathcal{I}_k\}$. The set of inflated obstacle boundary cells generated at step k that belong to the same $A(u)$ where $u \in V_{k-1}$ is defined as $\xi^u_k = \{c : c \in \Delta\mathcal{I}^{B,new}_k$ s.t. $\mathcal{V}(c) = u \in V_{k-1}\}$. Then, the set Ξ_k is defined as the partition of $\mathcal{I}^{B,new}_k$ using all the possible ξ^u_k (i.e. $\mathcal{I}^{B,new}_k = \bigcup_{\xi^u_k \in \Xi_k} \xi^u_k$).

Suppose $\xi^{u,v}_k$ denotes the set of inflated obstacle boundary cells belonging to a new safe-patch $A(v)$, $v \in V_k$ generated after splitting $A(u)$, $u \in V_{k-1}$ at step k. Then, Lemma 1 presents the operation required to identify a safe-patch resulted from splitting an old safe-patch.

Lemma 1. *If function* UPDATEV *is executed with $c \in \xi^{u,v}_k$ and $\mathcal{V}(c)$ as inputs and generates the safe-patch $A(\tilde{v})$, $\tilde{v} \in V_k$, then $A(\tilde{v}) \equiv A(v)$.*

Proof. Consider that $A(\tilde{v}) \neq A(v)$. Then $\nexists\, \tilde{c}$ s.t. $(\tilde{c} \in A(\tilde{v})$ and $\tilde{c} \in A(v))$. Since $c \in \xi^{u,v}_k$, then $c \in A(v)$. Also $c \in A(\tilde{v})$, thus leads to a contradiction. $\square$

Therefore, executing UPDATEV for a cell $c \in \xi^{u,v}_k$ for each v, and for all nodes u such that $\exists\, \xi^u_k \in \Xi_k$, generates the set $\hat{\Upsilon}_k$. Fig. 10 illustrates the use of inflated obstacle boundary cell sets ξ^{u,v_1}_k, ξ^{u,v_2}_k to split safe patch $A(u)$ to extract safe patches $A(v_1)$ and $A(v_2)$. The arrows indicate the wavefront propagation of flood-fill operations initiated from ξ^{u,v_1}_k and ξ^{u,v_2}_k.

5.3.3. *Removing split nodes Υ^s_k and adding nodes in $\hat{\Upsilon}_k$ to $\mathcal{T}_{k-1}$*

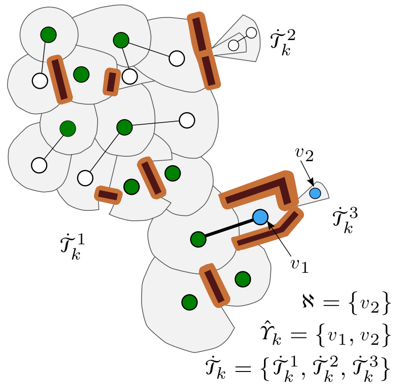
Each removal of a node in Υ^s_k and adding its substitute nodes in $\hat{\Upsilon}_k$ to the spanning trees require remapping of parent–child links in the spanning trees. The following set of rules ensure the retention of information of connected components even after the safe-patch splitting operations. Let $u \in \Upsilon^s_k$ and v be any resulting node after splitting safe-patch $A(u)$, and r^u and r^v are the two root nodes of the spanning trees associated with u and v. Then,

1. If $\exists$ some $n \in \mathcal{N}^v$ s.t. n is visited $\Rightarrow$ *parent*$[v] = n$.
2. If $\nexists\, n \in \mathcal{N}^v$ s.t. n is visited, and for some $n \in \mathcal{N}^v$ *parent*$[u] = n$ and *parent*$[v] = NIL \Rightarrow$ *parent*$[v] = n$.
3. If $\nexists\, n \in \mathcal{N}^v$ s.t. (n is visited or *parent*$[u] = n) \Rightarrow$ *parent*$[v] = n'$ for any $n' \in \mathcal{N}^v$ that maintains tree properties.
4. If $\nexists\, n \in \mathcal{N}^v$ s.t. n is visited, and for some $n \in \mathcal{N}^v$*parent*$[n] = u \Rightarrow$ *parent*$[n] = v$.
5. If for some n *parent*$[n] = u$ and $\nexists\, v$ s.t. $n \in \mathcal{N}^v \Rightarrow$ add n to $\aleph_k$.
6. If $r^v \neq v$ and $r^v \neq r^u \Rightarrow$ add r^v to $\aleph_k$.

Rules 1, 2 and 3 are mutually exclusive to each other and ensure that the new node v is connected to some tree if a connection is available. Rule 1 connects v to the spanning tree representing reachable connected component. Rule 2 connects v to the same spanning tree that u was a member of using the same parent node as u as illustrated in Fig. 11. Rule 3 connects new node v to a tree using a different parent node. It may or may not connect v to the same spanning tree of u. Rule 4 is used to redirect parent links of child nodes of u to node v. If such re-directions cannot be found, the sub-trees originating from these child nodes get disconnected from existing spanning trees. Hence these child nodes are stored in the set $\aleph_k$ using Rule 5 to ensure the connectivity information of the aforementioned sub-trees are not lost. If a parent link of a child node of u is redirected to v using Rule 4 and v is connected to a different tree (i.e. $r^u \neq r^v$) based on Rule 3 or not connected to a tree at all, possible connectivity information between nodes in the tree containing v and the tree that contained node u could be lost due to the local nature of these operations. Hence, the root of the tree containing v, r^v, is stored in $\aleph_k$ using Rule 6. The trees stored in $\aleph_k$ are enumerated later (Section 5.3.8) to check for possible global connectivity among different trees and to generate the globally consistent spanning tree set $\mathcal{T}_k$.

5.3.4. *Extracting new safe-patches to generate Υ'_k*

The set of newly added safe cells $\mathcal{S}^{new}_k = \{\mathcal{S}_k \cap U_k\}$ need to be abstracted into the graph G_k by generating safe-patches. Let $\mathbf{S}_k$ be the partitioning of $\mathcal{S}^{new}_k$ to safe-patches. Then Lemma 2 presents the operation to extract a safe-patch from $\mathcal{S}_k$.



$\dot{\mathcal{T}}_k^2$
v_2
$\dot{\mathcal{T}}_k^3$
$\dot{\mathcal{T}}_k^1$
v_1
$\aleph = \{v_2\}$
$\hat{\Upsilon}_k = \{v_1, v_2\}$
$\dot{\mathcal{T}}_k = \{\dot{\mathcal{T}}_k^1, \dot{\mathcal{T}}_k^2, \dot{\mathcal{T}}_k^3\}$

Fig. 11. Splitting a safe patch to new safe patches $A(v_1)$ and $A(v_2)$. $A(v_1)$ is connected to another node following Rule 2.

Lemma 2. *Given a cell $c \in S^i$, where $S^i \in \mathbf{S}_k$, executing* UPDATEV *on c extracts the safe-patch S^i from $\mathcal{S}_k^{new}$ and generates its associated node. (Proof is similar to Lemma 1).*

Therefore, executing UPDATEV for one cell $c \in S^i$ for each $S^i \in \mathbf{S}_k$ generates the set Υ_k'.

5.3.5. *Attaching new nodes in Υ_k' to existing trees in $\dot{\mathcal{T}}_k$*

Let $\Upsilon_k^V \subseteq \Upsilon_k'$ be the set of visited new nodes. Then, consider the set $\Upsilon_k = \Upsilon_k' \setminus \Upsilon_k^V$ of newly generated nodes that are not marked as visited by the robot. The following theorem states how a node from Υ_k can be connected to a spanning tree in $\dot{\mathcal{T}}_k$.

Theorem 1. *Consider a node $v \in \Upsilon_k$ such that v belongs to some connected component C_k^i in G_k and let $\dot{\mathcal{T}}_k^i \subseteq \dot{\mathcal{T}}_k$ be the collection of trees that contain the node set $C_k^i \cap \{V_{k-1} \cup \hat{\Upsilon}_k \setminus \Upsilon_k^s\}$. Then, $\exists$ a node $n \in \mathcal{N}^v$ s.t. $n \in nodes[\dot{\mathcal{T}}_k^i]$.*

Proof. Consider, for the purpose of contradiction, that $\nexists$ a node $n \in \mathcal{N}^v$ s.t. $n \in nodes[\dot{\mathcal{T}}_k^i]$. Therefore, $\forall n \in \mathcal{N}^v$, $n \notin nodes[\dot{\mathcal{T}}_k^i]$. Since v belongs to C_k^i, all the neighboring nodes of v also belong to C_k^i. Since for all $n \in \mathcal{N}^v$, $n \notin nodes[\dot{\mathcal{T}}_k^i] \Rightarrow n \in \Upsilon_k$. Therefore, for any $c_1 \in A(n)$ and $c_2 \in A(v)$, $us(c_1) = us(c_2) = k$, since both $n, v \in \Upsilon_k$ for any $n \in \mathcal{N}^v$. Then according to Proposition 2, n and v cannot be two distinct nodes, which leads to a contradiction. □

Hence, according to the above theorem, each newly generated non-visited node $v \in \Upsilon_k$ that gets added to an existing connected component from G_{k-1} can be attached to a node in an existing spanning tree in $\dot{\mathcal{T}}_k$. For such nodes, a connecting edge can be found during the generation of these nodes. The set Φ_k is defined to store all such edges, one for each node in Υ_k. Fig. 12 illustrates connecting a new safe patch $A(v_3) \in \Upsilon_k$ using an edge (v_2, v_3) in Φ_k.

5.3.6. *Updating information about visited nodes*

A node in V_k can be marked visited according to two scenarios. In the first scenario, for any newly generated $v \in \{\Upsilon_k' \cup \hat{\Upsilon}_k\}$, if the robot's current position grid cell c_k^* is in $A(v)$, then v is marked visited by directly adding it to the spanning tree representing the reachable connected components. The second scenario involves splitting a visited safe-patch due to obstacle information. The following provides a rule to mark such newly generated nodes as visited.

Lemma 3. *Let $\mathcal{P}_k$ be the trajectory of the robot so far. Let $A(u)$ be a safe-patch with $u \in V_{k-1}$ being a visited node. Consider, splitting $A(u)$ to multiple safe-patches represented by $\Omega = \{v_1, v_2, \ldots\}$ due to new obstacle information. Then,* (a) *Splitting generates only one visited node v_i* (b) $\{\mathcal{P}_k \cap A(u) = \mathcal{P}_k \cap A(v_i)\}$.

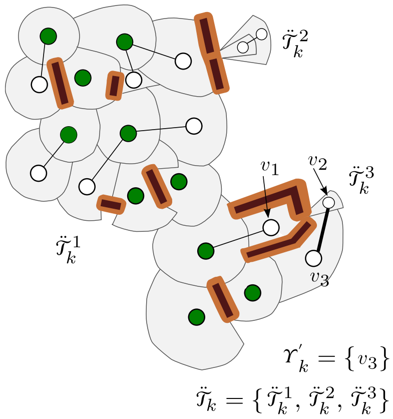


Fig. 12. Connecting a new node in Υ_k to an existing node from trees $\dot{\mathcal{T}}_k$. Resultant trees $\ddot{\mathcal{T}}_k^1, \ddot{\mathcal{T}}_k^2$ belong to the same connected component.

Proof. (a) Consider, for the purpose of contradiction, that splitting generates more than one safe-patch that is marked visited. Then $\exists$ $v_1, v_2 \in \Omega$ s.t. $A(v_1), A(v_2)$ are two visited safe-patches generated by splitting $A(u)$. Since, $A(v_1)$ and $A(v_2)$ are not connected, $\nexists$ a path from any $c_1 \in A(v_1)$ to any $c_2 \in A(v_2)$. Since, both v_1, v_2 are assumed visited, $\{\mathcal{P}_k \cap A(v_1)\} \subseteq A(v_1)$ and $\{\mathcal{P}_k \cap A(v_2)\} \subseteq A(v_2)$. Hence, $\nexists$ a path from $\forall c_1 \in \{\mathcal{P}_k \cap A(v_1)\}$ to $\forall c_2 \in \{\mathcal{P}_k \cap A(v_2)\}$, this leads to $\mathcal{P}_k$ is not connected, which is a contradiction.

(b) Let $\mathcal{I}_k^u \in \mathcal{I}_k$ be the set of new inflated obstacle grid cell information that triggered the splitting of $A(u)$. Then, $A(u) = \{\bigcup_k A(v_k)\} \cup \mathcal{I}_k^u$. Hence, $\mathcal{P}_k \cap A(u) = \mathcal{P}_k \cap \{\bigcup_i A(v_i)\} \cup \mathcal{I}_k^u = \{\bigcup_i \{\mathcal{P}_k \cap A(v_i)\}\} \cup \{\mathcal{P}_k \cap \mathcal{I}_k^u\}$. Since $\mathcal{I}_k^u \cap \mathcal{S}_k = \phi$, $\mathcal{P}_k \cap \mathcal{I}_k^u = \phi$. From (a), $\exists!$ $v_i \in \Omega$ s.t. v_i is visited, therefore, $\mathcal{P}_k \cap A(v_i) \neq \phi$ and $\mathcal{P}_k \cap A(v_j) = \phi \; \forall \, j \neq i$. Therefore $\mathcal{P}_k \cap A(u) = \mathcal{P}_k \cap A(v_i)$. □

Theorem 2. *Let $p \in \{A(u) \cap \mathcal{P}_k\}$ be a grid cell associated with visited node $u \in V_{k-1}$. Consider that $A(u)$ gets split to disjoint safe-patches represented by $\Omega = \{v_1, v_2, \ldots\}$ due to obstacle information. Then, if $p \in A(v_j)$ for some $v_j \in \Omega$, then $A(v_j)$ is the visited safe-patch generated by splitting $A(u)$.*

Proof. From Lemma 3, $A(u) \cap \mathcal{P}_k = A(v_i) \cap \mathcal{P}_k$ and v_i is the only node that is marked visited after splitting u. Since, $p \in \{A(u) \cap \mathcal{P}_k\} \Rightarrow p \in \{A(v_i) \cap \mathcal{P}_k\}$ and $\{A(v_i) \cap \mathcal{P}_k\} \subseteq A(v_i)$, it implies that $p \in A(v_i)$. If, $p \in A(v_j)$, then it implies that $A(v_j) \equiv A(v_i)$, hence v_j is the node representing the visited safe-patch generated by splitting $A(u)$. □

Due to the results of Lemma 3 and Theorem 2, associating a single cell $p \in \{A(u) \cap \mathcal{P}_k\}$ with each node u is sufficient to identify the node v that becomes the only visited node when $A(u)$ is split.

5.3.7. *Representing isolated nodes as separate spanning trees*

Given the set $\Upsilon_k^I = \{\Upsilon_k \setminus nodes[\ddot{\mathcal{T}}_k]\}$ of nodes that represent the newly generated nodes which are not connected to any spanning tree in $\ddot{\mathcal{T}}_k$, it can be easily deduced that for all $v \in \Upsilon_k^I$, $\mathcal{N}^v = \phi$, (i.e. all nodes v are isolated nodes with no neighboring nodes). Hence, they can be added to $\ddot{\mathcal{T}}_k$ trivially by making them the roots of their own spanning trees.

5.3.8. *Merging trees in $\ddot{\mathcal{T}}_k$ whose nodes belong to the same connected component*

New nodes in $\hat{\Upsilon}_k$ and Υ_k' are connected to one of neighboring nodes greedily and do not guarantee the connectedness of the connected components in G_k. Therefore, some of the trees in $\ddot{\mathcal{T}}_k$ represent only subsets of nodes in connected components of G_k as

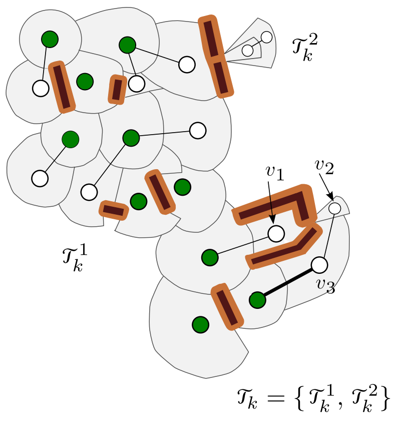


Fig. 13. Connecting the two trees $\ddot{\mathcal{T}}_k^1$ and $\ddot{\mathcal{T}}_k^3$ through node v_3.

illustrated by $\ddot{\mathcal{T}}_k^1$ and $\ddot{\mathcal{T}}_k^3$ in Fig. 12. Therefore, trees in $\ddot{\mathcal{T}}_k$ must be checked for their connectedness and merged to generate the final set of spanning trees $\mathcal{T}_k$.

Consider $v \in \Upsilon'_k$ to be a node attached to tree $\dot{\mathcal{T}}_k^i$. Any new connectivity of the corresponding tree $\ddot{\mathcal{T}}_k^i$ to another spanning tree $\ddot{\mathcal{T}}_k^j \in \ddot{\mathcal{T}}_k$ through v can be identified by checking the adjacency information $\Lambda(v)$ as depicted in Fig. 13. $\Lambda(v)$ is defined as $\Lambda(v) = \{\tilde{v} \in \mathcal{N}^v$ s.t. $parent[v] \neq \tilde{v}$ and $\tilde{v} \notin child\text{-}nodes[v]$ in the spanning tree $\mathcal{T}_k^i$ where $v \in nodes[\mathcal{T}_k^i]\}$.

A spanning tree $\mathcal{T}_{k-1}^i \in \mathcal{T}_{k-1}$ could be split to several sub-trees when a node is replaced by several non-connected nodes during a splitting of a safe-patch. However the resulting sub-trees could be connected through some other nodes which are not in Υ'_k. Therefore these sub-trees need to be checked for merging with other trees through nodes not in Υ'_k.

Theorem 3. *Assume $\mathcal{T}_{k-1}$ contains the spanning trees representing the connected components of G_k. Let $\dot{\mathcal{T}}_k^i$ and $\dot{\mathcal{T}}_k^j$ be two distinct trees in $\dot{\mathcal{T}}_k$ s.t. $\forall\ v \in \Upsilon'_k$, $\mathcal{N}^v \cap \{nodes[\dot{\mathcal{T}}_k^i] \cup nodes[\dot{\mathcal{T}}_k^j]\} = \phi$ and $\exists$ some $e = (v_k, v_l) \in E_k$ s.t. $v_k \in nodes[\dot{\mathcal{T}}_k^i]$ and $v_l \in nodes[\dot{\mathcal{T}}_k^j]$. Then at least one of the nodes r^i, r^j is in the set $\aleph_k$, where r^i and r^j are the root nodes of $\dot{\mathcal{T}}_k^i$ and $\dot{\mathcal{T}}_k^j$ respectively.*

Proof. Since none of the nodes in Υ'_k connect $\dot{\mathcal{T}}_k^i$, $\dot{\mathcal{T}}_k^j$ together, then both v_k and v_l must be in $\{V_{k-1} \cup \hat{\Upsilon}_k \setminus \Upsilon_k^s\}$. From Lemma 1 it can be deduced that at most only one of v_k, v_l can be in $\hat{\Upsilon}_k$. (a) If both v_k and $v_l \in \{V_{k-1} \setminus \{\hat{\Upsilon}_k \cup \Upsilon_k^s\}\}$, then v_k, v_l must be in the same tree $\mathcal{T}_{k-1}^m \in \mathcal{T}_{k-1}$, based on the assumption of $\mathcal{T}_{k-1}$ representing all connected components of G_{k-1}. Then v_k and v_l can be nodes of distinct $\dot{\mathcal{T}}_k^i$, $\dot{\mathcal{T}}_k^j$ only through safe-patch splitting. Since the two trees are distinct, according to Rules 5 and 6 in Section 5.3.2, at least one of r^i or r^j must be in $\aleph_k$. (b) Without lost of generality, assuming only $v_k \in \hat{\Upsilon}_k$, then $r^i \neq r^j$, therefore according to Rule 6 in Section 5.3.2, r^i must be in $\aleph_k$. □

Theorem 3 leads to the result that, if two trees in $\ddot{\mathcal{T}}_k$ are directly connected through two nodes in $\{V_{k-1} \cup \hat{\Upsilon}_k \setminus \Upsilon_k^s\}$, then the connecting edge can be retrieved by examining the trees stored in the set $\aleph_k$. Therefore, iterating through the nodes in Υ'_k and nodes of trees in $\aleph_k$ to check for possible tree connectedness ensures generation of the set of spanning trees $\mathcal{T}_k$ representing connected components of G_k, given $\mathcal{T}_{k-1}$. Given all the trees in $\mathcal{T}_1$ at first map update step are valid spanning trees representing connected components of G_1, it is deduced that the provided operations incrementally maintain the spanning trees representing connected components of G_k for each k and can be used to check the reachability of the extracted mapped-free-space boundary cells.

Table 2
Data structures used during implementation.

$\Lambda(v)$	The adjacent nodes of v that are neither child nor parent nodes of v
Ξ_k	Partition of inflated obstacle boundary cells generated at step k according to membership of safe-patches represented by V_{k-1}
$\aleph_k$	Set of root nodes of sub-trees that got disconnected during safe-patch splitting
Φ_k	The set of edges connecting new nodes in Υ'_k to existing trees

Algorithm 2 Update-Modified-Nodes(Ξ_k)

```
1: for all $\xi \in \Xi_k$ do
2:   $u \leftarrow \mathcal{V}(\xi[1])$
3:   $r^u \leftarrow$ GetRoot($u$)
4:   for $c \in \xi$ do
5:     if $\mathcal{V}(c) \neq u$ OR $c \notin \mathcal{S}_k$ then
6:       continue
7:     $(v, \mathcal{N}^v) \leftarrow$ UpdateV($c, u$)
8:     for all $n \in \mathcal{N}^v$ do
9:       if $visited[n] =$ TRUE AND $parent[v] =$ NIL then
10:        make $n$ the parent of $v$
11:      else if $parent[n] = u$ then
12:        make $v$ parent of $n$
13:      else
14:        make $v$ and $n$ adjacent nodes
15:        if $parent[u] = n$ then
16:          $e = (n, v)$
17:    if $parent[v] =$ NIL then
18:      if $e \neq$ NIL then
19:        make $e[1]$ parent of $e[2] = v$
20:      else
21:        connect $v$ to a node $n' \in \Lambda(v)$ s.t. edge $(n', v)$ is not
           in a cycle
22:    if ($visited[u] =$ TRUE AND $\mathcal{V}(RPC[u]) = v$) OR $\mathcal{V}(c_k^*) = v$
       then
23:      $visited[v] \leftarrow$ TRUE
24:      $RPC[v] \leftarrow c_k^*$
25:      if $parent[v] = n$ AND $visited[n] =$ FALSE then
26:        make $v$ the parent of $n$
27:      make $v$ the parent of $v$
28:    else
29:      $r^v \leftarrow$ GetRoot($v$)
30:      if $r^v \neq r^u$ then
31:        $\aleph_k \leftarrow \aleph_k \cup \{r^v\}$
32:  for each old child node $n$ of $u$ do
33:    if $parent[n] = u$ then
34:      $\aleph_k \leftarrow \aleph_k \cup \{n\}$
35:  RemoveNode($u$)
```

6. Implementation details

6.1. Algorithms for maintaining $\mathcal{T}_k$

The algorithms based on the formalization of update operations in Section 5.3 are provided below. Given the algorithms are provided in sufficient detail and in the interest of brevity, only the important operations are highlighted. Table 2 summarizes the important data structures used during the implementation of reachability maintenance algorithms.

Algorithm 2 details the safe-patch splitting operation based on new obstacle information. The input to the algorithm is the set Ξ_k. Each inflated obstacle boundary cluster ξ_k^u belonging to a node $u \in V_{k-1}$ is iterated in first loop. The check in line 5 ensures that UpdateV is executed only once for each $v \in \hat{\Upsilon}_k$ according to Lemma 1 thus generating $\hat{\Upsilon}_k$. Node connectivity Rules 1, 2, 3 and 4 given in Section 5.3.2 are executed in lines 9–10, 18–19, 20–21

Algorithm 3 Insert-New-Nodes($\mathcal{S}_k^{new}$)

```
1:  $\Upsilon_k' \leftarrow \phi$
2:  for all $c \in \mathcal{S}_k^{new}$ do
3:    if $\mathcal{V}(c) = $ NIL then
4:      $(v, \mathcal{N}^v) \leftarrow$ UpdateV$(c,$ NIL$)$
5:      $e \leftarrow \phi$
6:      $\Upsilon_k' \leftarrow \Upsilon_k' \cup \{v\}$
7:      for all $n \in \mathcal{N}^v$ do
8:        if $e = \phi$ then
9:          $e \leftarrow (n, v)$
10:       else
11:         make $n$ and $v$ adjacent nodes
12:     if $\mathcal{V}(c_k^*) = v$ then
13:       $visited[v] \leftarrow$ TRUE
14:       $RPC[v] \leftarrow c_k^*$
15:       make $v$ the parent of $v$
16:       make $e[1]$ and $v$ adjacent nodes
17:     else
18:       $\Phi_k \leftarrow \Phi_k \cup \{e\}$
```

and 11–12 respectively. Rule 5 and 6 are executed in lines 33–34 and 30–31 to populate the set $\aleph_k$. For each visited node v, variable *RPC* (Robot Position Cell) is updated in line 24 to store a grid cell in $\mathcal{P}_k \cap A(v)$ according to Theorem 2 to maintain information about visited nodes. At the end of the execution of Algorithm 2, the split nodes in Υ_k^s are removed from the spanning trees in $\mathcal{T}_{k-1}$ and their substituting nodes $\hat{\Upsilon}_k$ are generated and added to the trees based on the Rules 1–6 to generate the set of trees $\dot{\mathcal{T}}_k$. Considering δm to be the map resolution and δd to be the avg. change in robot's position between to update steps, the maximum possible safe-patch size in grid cells is given by $2\delta dR_{max}/\delta m$. Then the worst case complexity of processing updated safe-patches due to obstacle information is given by $\mathcal{O}(2|\Xi_k|\delta dR_{max}/\delta m)$. The complexity of executing the Rules 1–6 in Section 5.3.2 is given by $\mathcal{O}(deg_{max}(G_k))$ where $deg_{max}(G_k)$ is the maximum node degree in graph G_k and $deg_{max}(G_k) \ll \delta dR_{max}/\delta m$. Hence the complexity of Algorithm 2 is given by $\mathcal{O}(2|\Xi_k|\delta dR_{max}/\delta m)$.

Algorithm 3 details generation of safe-patches with new safe cell data. The input to the algorithm is $\mathcal{S}_k^{new}$. The check in line 3 ensures that UpdateV is executed only once for each safe-patch $A(v)$, $v \in \Upsilon_k'$ according to Lemma 2 to generate the set Υ_k'. Set Φ_k is populated for each non-visited node in line 18 according to Theorem 1 by selecting any adjacent node greedily (line 9). The complexity of generating new safe patches is linear in the number of new safe cells hence given by $\mathcal{O}(|\mathcal{S}_k^{new}|)$. Algorithm 3 uses the resulting Φ_k from Algorithm 3 to attach non-visited new nodes in Υ_k to existing trees in $\dot{\mathcal{T}}_k$ and its complexity is given by $\mathcal{O}(|\Phi_k|)$. These node insertion operations transform the set $\dot{\mathcal{T}}_k$ of trees to $\ddot{\mathcal{T}}_k$ according to the work-flow given in Fig. 8.

Algorithm 5 details the first type of tree merging operation conducted on $\ddot{\mathcal{T}}_k$ as formulated in Section 5.3.8. The input to the algorithm is Υ_k'. For each node in $v \in \Upsilon_k'$, its adjacent nodes $\Lambda(v)$ are examined for possible connections with other trees in lines 5, 7 and 9. If one of the trees to be merged is currently representing nodes in reachable connected component (i.e. root node is ν), then the other tree is merged as a sub-tree. If none of the trees are currently representing nodes in reachable connected component, then the shorter of the two trees is merged with the other using the height check in line 10. Two trees are merged

Algorithm 4 Attach-New-Nodes-To-Spanning-Trees(Φ_k)

```
1: for $i \leftarrow 1..|\Phi_k|$ do
2:   access $\Phi_k[i]$ as edge $e = (n, v)$
3:   make $n$ parent of $v$
```

Algorithm 5 Connect-Spanning-Trees-Through-New-Nodes(Υ_k')

```
1:  for all $v \in \Upsilon_k'$ do
2:    $(r^v, h^v) \leftarrow$ GetRoot$(v)$
3:    for all $a \in \Lambda(v)$ do
4:      $(r^a, h^a) \leftarrow$ GetRoot$(a)$
5:      if $r^v \equiv \nu$ and $r^v \neq r^a$ then
6:        MergeToTree$(v, a)$
7:      else if $r^a \equiv \nu$ and $r^a \neq r^v$ then
8:        MergeToTree$(a, v)$
9:      else if $r^v \neq r^a$ then
10:       if $h^a > h^v$ then
11:         MergeToTree$(a, v)$
12:       else
13:         MergeToTree$(v, a)$
MergeToTree(a, b)
1: Make $b$ the root of it is tree
2: Make $a$ parent of $b$
```

Algorithm 6 Connect-Spanning-Trees-Through-Old-Nodes($\aleph_k$)

```
1:  for all $r \in \aleph_k$ do
2:    $\mathbf{V} \leftarrow \mathbf{V} \cup \{$ExtractNodes$(r)\}$
3:  for $i \leftarrow 1 \ldots |\mathbf{V}|$ do
4:    $\mathcal{R} \leftarrow \phi$
5:    for all $v \in \mathbf{V}[i]$ do
6:      for all $a \in \Lambda(v)$ do
7:        $(r^a, h^a) \leftarrow$ GetRoot$(a)$
8:        if $r^a \neq \aleph_k[i]$ and $\nexists e \in \mathcal{R}$ s.t. $e$ connects trees with roots
          $r^v$ & $r^a$ then
9:          $\mathcal{R} \leftarrow \mathcal{R} \cup \{(v, a)\}$
10:   for all $e \in \mathcal{R}$ do
11:     MergeToTree$(e[1], e[2])$
```

through nodes a and b using function MergeToTree by re-rooting one of the trees. Both the root extraction operations (in lines 2, 4) and tree merging operations have complexities of $\mathcal{O}(s_u + 1)$ based on the maximum depths defined in Sections 5.2.1 and 5.2.2. Since maximum branching factor of trees is less than $deg_{max}(G_k)$, the complexity of Algorithm 5 is given by $\mathcal{O}(|\Upsilon_k'|deg_{max}(G_k)(s_u + 1))$.

Following the result of Theorem 3, the Algorithm 6 utilizes the set $\aleph_k$ for the second type of tree merging operation formulated in Section 5.3.8. First the algorithm enumerates all the nodes of trees in $\aleph_k$ (line 2). Then each node is examined for possible connection of two distinct trees. Check in line 8 ensures only one such edge is stored in set $\mathcal{R}$ for each unique tree pair. Finally the edges in $\mathcal{R}$ are iterated to merge all the connected trees. Similar to Algorithm 5, the complexity of Algorithm 6 is given by $\mathcal{O}(|V|deg_{max}(G_k)(s_u+1))$. At the end of execution of Algorithms 5 and 6, the set of trees in $\ddot{\mathcal{T}}_k$ are transformed to the set $\mathcal{T}_k$ of spanning trees representing the connected components of graph G_k.

6.2. Contour database

The database that incrementally maintains the information about newly generated map boundary contours and disappeared contours is described in this section. The data structure that implements the database must be able to support three key operations efficiently. These are, deletion of disappeared contours, insertion of newly identified contours and extraction of the contours. All three of these operations require efficient access to spatial data of contours. Therefore spatial data structures are considered for the implementation of the contour database.

While there are spatial data structures to store and query information about properly defined line segments such as interval trees

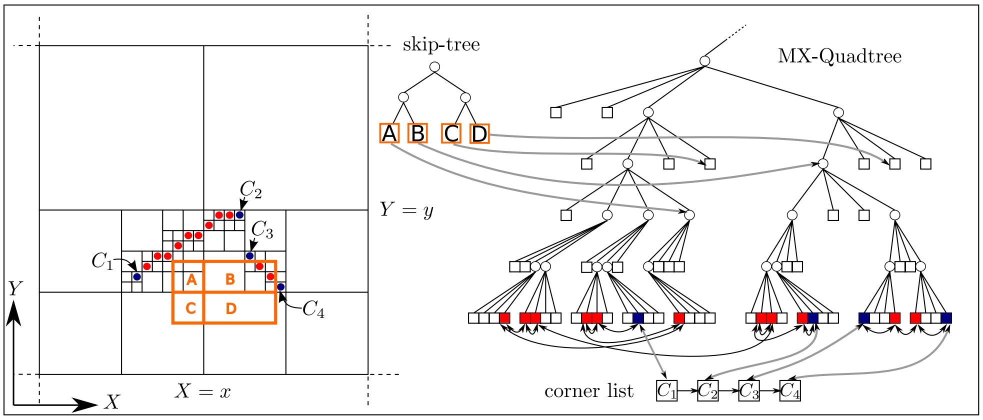


Fig. 14. The set of grid information, of a larger grid (left) and the Linked MX-Quadtree (L-MX-Quadtree) that represents the cell data (right). All the leaf cells in the MX-Quadtree that represent valid contour cells are chained together according to their contour membership. The linked list of corner cells contains links to the nodes representing the corner cells of stored contours. Leaf nodes of skip-tree, to the left of the quadtree, contain links to the nodes that represent the smallest quadrants that enclose the respective sub-rectangles represented by the leaf nodes of skip-tree.

and segment trees [37], these are not suitable to maintain non-parametrized contour information of map boundary cells. Therefore, a spatial data structure that is based on MX-Quadtree [38] which is capable of maintaining all the cell information of valid contours is presented. Reader is referred to [38] for a detailed explanation on Quadtrees and related data-structures. While MX-Quadtree provides an easy way of representing all the individual cells of the valid mapped-space contours, MX-Quadtree in its current form is not an efficient data structure to store and manage large amount of contour information. This is due to the leaves on MX-Quadtrees always corresponding to 1×1 grid cells, resulting in traversing a large number of intermediate nodes to access the required leaves. Therefore a skip-list [39] inspired modification is introduced to generate the Linked-MX-Quadtree (L-MX-Quadtree) to efficiently represent map boundary contour information and is described next.

6.2.1. *Linked MX-Quadtree (L-MX-Quadtree) to store contours*

In order to preserve contour information during the insertion of individual grid cells to the MX-Quadtree, the leaves in the MX-Quadtree which represent the grid cells that are next to each other in the contour are linked with each other. In addition, the two corners of each contour are stored in a list called *corner list*. Each list item links to its corresponding leaf node in the MX-Quadtree as depicted in the linked list below the quadtree in Fig. 14 and is similar to the operation of a skip-list [39]. Associating a corner-list item with a leaf node in L-MX-Quadtree provides an easy check for detecting corner nodes of stored contours.

6.2.2. *Retrieving frontier cells*

Since all boundary cells that belong to a particular contour bound the same free area, they share the same reachability value. Therefore checking the reachability of a single cell in a contour is sufficient to check whether the contour is reachable or not so that it can be retrieved as a frontier contour. Once the reachability of the contour is established, the corner-list is used to index into nodes representing corner cells of the contour. The rest of the cells of the contour can be retrieved by following the chained links among the nodes. When the other corner of the contour is encountered, the retrieving of the contour is terminated and the associated corner node is marked off in the corner-list so that it is not used again to retrieve the same contour twice. Considering $C_{\mathcal{B},k}$ be the set of boundary contours stored in the tree, all these contours can be accessed through the *corner-list* with a complexity of $\mathcal{O}(|C_{\mathcal{B},k}|)$. The reachability of each contour can be evaluated relatively in constant time. Since only the frontier cells are retrieved, only $\mathcal{O}(|\Gamma_k|)$ time is needed to enumerate the reachable contours where Γ_k is the set of frontier cells at step k. Therefore the complexity of retrieving all the frontier cells is given by $\mathcal{O}(|\Gamma_k|) + \mathcal{O}(|C_{\mathcal{B},k}|)$.

6.2.3. *Improving indexing complexity*

While the corner list improves the efficiency of retrieving all the contour information from the data structure, it does not improve the time needed to index into a specified contour cell, an operation that occurs frequently during the updating of the contour database. Since the information about the local area on which all the map updates are conducted is known (i.e. U_k), this information can be used to skip most of the intermediate nodes during search operations of the MX-Quadtree. A skip-tree is generated at each map update step k using the bounding box information of the set U_k to achieve this task. The area of the bounding box is divided to a maximum of four rectangles based on quadrant margins from the quadtree, an operation which is similar to area subdivisions in kd-trees [40,37]. An example of a subdivision of a bounding box is depicted by the orange/bold rectangles in the grid of Fig. 14. The orange rectangle is the bounding box of U_k and it is divided to four rectangles A, B, C, D based on the two quadrant boundary lines at $X = x$ and $Y = y$. Each leaf node in the skip-tree that represent these sub-rectangles links to the smallest quadrant in the quadtree that fully encloses the sub-rectangle in question. Therefore search operations involving cells within the bounding box, are able to skip large number of intermediate nodes from the root of quadtree by retrieving information about a closer ancestral quadrant from the skip-tree.

The generation of the skip-tree is conducted during the first indexing operation for each update step. Hence, the complexity of this initial indexing operation becomes $\mathcal{O}(\log_2 s_m)$ where s_m is the dimension of the grid. All the subsequent indexing operations have the complexity of $\mathcal{O}(\log_2 d_k^u)$ where d_k^u is the dimension of the smallest quadrant enclosing the bounding box of U_k. Considering $d_k^b = \max(\mathit{width}(U_k), \mathit{height}(U_k))$ be the dimension of the bounding box of U_k, the inequality $d_k^u < 2d_k^b$ leads to the complexity of subsequent indexing operations to be $\mathcal{O}(\log_2 2d_k^b)$. ($\ll \mathcal{O}(\log_2 s_m)$ given $2d_k^b \ll s_m$).

6.2.4. *Deleting contours*

The set of disappeared boundary cells that is to be deleted during current update step is stored in $\mathcal{B}_k^{del}$. Since all the cells in $\mathcal{B}_k^{del}$ are parts of contours, the contour information stored in L-MX-Quadtree is utilized to improve the efficiency of the deletion operation as illustrated in Algorithm 7. In line 2, the node in the

Algorithm 7 Delete-Contours($\mathcal{B}_k^{del}$)

```
1:  for all $c \in \mathcal{B}_k^{del}$ do
2:      Node $n \leftarrow$ FindQTNode($c$)
3:      Node $nn \leftarrow$ next[$n$]
4:      Node $pn \leftarrow$ prev[$n$]
5:      DeleteNode($n$)
6:      $\mathcal{B}_k^{del} \leftarrow \mathcal{B}_k^{del} \setminus \{\text{cell}[nn]\}$
7:      while $nn \in \mathcal{B}_k^{del}$ do
8:          Node $n \leftarrow$ next[$nn$]
9:          DeleteNode($nn$)
10:         $\mathcal{B}_k^{del} \leftarrow \mathcal{B}_k^{del} \setminus \{\text{cell}[nn]\}$
11:         $nn \leftarrow n$
12:     while $pn \in \mathcal{B}_k^{del}$ do
13:         Node $n \leftarrow$ prev[$pn$]
14:         DeleteNode($pn$)
15:         $\mathcal{B}_k^{del} \leftarrow \mathcal{B}_k^{del} \setminus \{\text{cell}[pn]\}$
16:         $pn \leftarrow n$
```

L-MX-Quadtree that represents a disappeared boundary cell is located. The skip-tree based search is used to efficiently retrieve the node. Once the node is found, its next and previous nodes in the contour are retrieved using the links among leaf nodes, lines (3–4). The contour is traversed in both directions from the seed nodes *nn* and *pn* to delete nodes until non-disappeared nodes are encountered, lines 7–11 and 12–16. The deletion operation slightly differs from standard quadtree deletion where collapsing of empty nodes is conducted only up to quadrant enclosing the bounding box of U_k. This is to ensure the validity of the links in the skip-tree. During the deletion step of a node, if it is a node representing a corner cell of a contour, the associated item in the corner-list is also deleted.

Considering $C_{\mathcal{B},k}^{del}$ to be the partial-contours being deleted from the contour database, the complexity of indexing to each of the deleted partial-contours is given by $\mathcal{O}(|C_{\mathcal{B},k}^{del}|\ \log_2(2d_k^b))$. Since each cell in $\mathcal{B}_k^{del}$ is deleted, the complexity of the deletion operations is given by $\mathcal{O}(|\mathcal{B}_k^{del}|\ \log_2(2d_k^b))$. Hence the total complexity of the deletion operation is $\mathcal{O}((|C_{\mathcal{B},k}^{del}| + |\mathcal{B}_k^{del}|)\ \log_2(2d_k^b))$.

6.2.5. Inserting contours

The newly appeared boundary cells $\mathcal{B}_k^{new}$ are added to the L-MX-Quadtree using the extracted contour information $C_{\mathcal{B},k}^{new}$. For each contour the nodes representing the constituting cells are inserted by first indexing to the smallest available quadrant enclosing the 1×1 quadrant representing the node, using the skip-tree. Then a repetitive sub-division is conducted until the corresponding 1×1 quadrant is created. Algorithm 8 details the important steps of the insertion procedure. In lines 2–4, all the cells of new contours are added to the L-MX-Quadtree and the nodes representing adjacent cells in the contours are linked together to preserve contour information. For each new contour if a corner cell c of contour has an adjacent valid old boundary cell c' (the check in line 7), then nodes representing c and c' are linked to merge the two contours (line 8). Since c' was previously a corner and now it is merged to another contour invalidating its "corner" status, the associated node is deleted from the corner list (line 9). If a corner cell c for a new contour does not have an adjacent old contour cell, c marks an actual corner of a contour. Therefore a corner node associated with c is added to the corner list (line 11).

Since each node representing a new boundary cell is inserted to the tree, the complexity of node insertions becomes $\mathcal{O}(|\mathcal{B}_k^{new}|\ \log_2(2d_k^b))$. The complexity of linking nodes corresponding to adjacent cells in contours is linear in the number of new boundary cells therefore already included in the earlier term.

Algorithm 8 Insert-Contours($C_{\mathcal{B},k}^{new}$)

```
1:  for all contour $C \in C_{\mathcal{B},k}^{new}$ do
2:      for all $c \in C$ do
3:          Node $n \leftarrow$ InsertNode($c$)
4:          Link previous and next node in chain
5:      $c_{start} \leftarrow C[1]$, $c_{end} \leftarrow C[end]$
6:      for all $c \in \{c_{start}, c_{end}\}$ do
7:          if $\exists\, c' \in \{\mathcal{B}_{k-1} \setminus \mathcal{B}_k^{del}\}$ s.t. $c' \in \mathcal{N}_8(c)$ then
8:              Link($node[c']$, $c$)
9:              Remove $node[c']$ from corner list
10:         else
11:             Insert $c$ to corner list
```

Since new corner-list items are added during the insertion process, the complexity of this step becomes $\mathcal{O}(|C_{\mathcal{B},k}^{new}|)$. Therefore, the total complexity of the contour insertion process is given by $\mathcal{O}(|\mathcal{B}_k^{new}|\ \log_2(2d_k^b)) + \mathcal{O}(|C_{\mathcal{B},k}^{new}|)$.

7. Experimental evaluation

Experiments are conducted to evaluate the efficiency and the accuracy of the proposed frontier generation strategy Safe and Reachable Frontier Detection (SRFD) using real world data and the results are compared with the traditional method of frontier generation (TRA), WFD, WFD-INC and OFD methods. The maps of the data sets used for evaluation are depicted in Fig. 15. Data for the IRL and ARRL environments are collected by tele-operating a Pioneer-3AT robot equipped with a Hokuyo 2D laser range finder. Data for FR101, CMU-NSH and Seattle-R environments were obtained from the Robotics Data Set Repository (Radish) [41]. The localization of the robot is achieved by executing a popular SLAM implementation GMapping [12]. We used our own implementation of occupancy grid mapping algorithm in [42] with a maximum usable sensor range set to 4 m following [18] to control the effect of ray scattering and specular reflection. This combined mapping approach is identical to systems that use a separate feature based SLAM component for localization and a separate grid based mapping component used for motion planning.

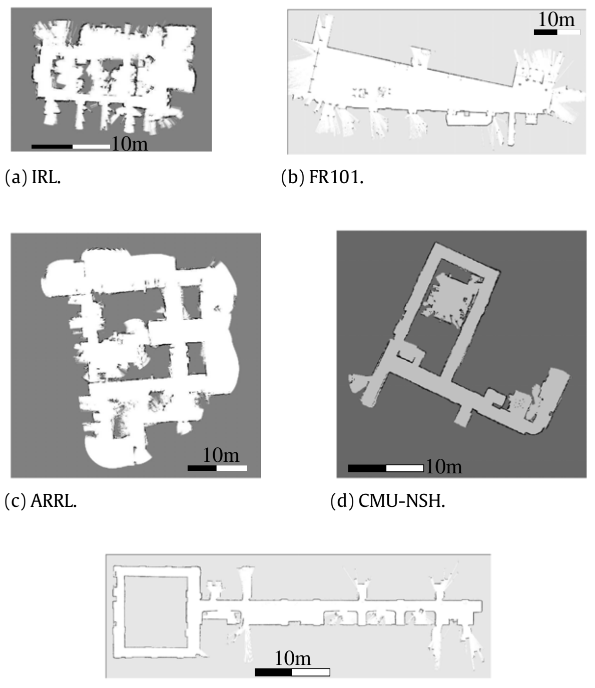
The occupancy grid maps produced in this manner are generally less accurate than integrated grid-based SLAM solutions. However this provides a valuable opportunity to evaluate the performance of the proposed algorithm in all types of environments including challenging environments with less accurate map information. In addition this avoids reinitialization of data structures for the WFD-INC method otherwise needed when "best" map changes in particle based systems. Hence WFD-INC and WFD-IP methods are identical in our evaluation and is referred to as $WFD_{INC/IP}$ here onwards. In all situations a map update is triggered when the robot has moved over 0.5 m or rotated over 0.1 rad. All methods used for comparisons are developed as ROS components [20] in C++ using default Linux and ROS libraries on a computer with Intel Core 2 Duo 3 GHz processor and 4 GB of RAM running Ubuntu 12.04.[1]

7.1. Comparison of execution time for frontier generation operations

7.1.1. Comparison with TRA method

The execution time to generate frontiers using SRFD method is first compared with traditional (TRA) method using IRL and ARRL environments. Grids with different sizes are used for IRL and ARRL environments respectively to evaluate the effect of the grid map

[1] The code used for experiments will be provided upon request, and will be released publicly in the future.



(a) IRL. (b) FR101. (c) ARRL. (d) CMU-NSH. (e) Seattle-R.

Fig. 15. Maps of data-sets used for experimental validation.

size on SRFD and TRA methods. For IRL environment grids with dimensions 800 × 800, 1000 × 1000 and 1200 × 1200 and for ARRL environment grid maps with dimensions 1200 × 1200 and 1600 × 1600 are considered with a fixed grid cell size of 5 cm/cell. For each map update step, the time taken to generate and publish the available frontier information to other components is recorded as execution time.

Fig. 16(a) and (b) compare the progression of the execution time along the mission between SRFD and TRA methods for IRL and ARRL environments respectively. Comparisons indicate that the execution time to generate and publish frontier information using the TRA approach increases with the map update step. And this increment closely follows the progression of the number of mapped cells in the environment (Fig. 17). As the number of mapped cells increase in the map, TRA method which processes the entire map has to process more grid cells, justifying the increase in the execution time. In addition, for the same environment with increasing grid map sizes the execution time increases for the traditional approach. This is again due to processing of the entire map. In comparison, for both environment types the execution time for generating frontiers using SRFD method is bounded across map update steps from the start to the end of the mission for all grid map sizes. This is directly due to processing of only local map information by SRFD.

7.1.2. *Comparison with other incremental methods*

SRFD method's execution time is also compared with WFD, OFD and $WFD_{INC/IP}$ methods. Figs. 18–20 report the execution times of each method averaged over a mapping mission on corresponding data-sets from start to end. Considering OFD and $WFD_{INC/IP}$ methods are similar in approach to SRFD, reachability detection by processing the full grid map is integrated to them in order to evaluate the effect of this operation on their execution times. The resultant methods are referred to as OFD^* and $WFD^*_{INC/IP}$ respectively and the associated execution times are also reported.

Different combinations of map dimensions (given in meters) and map resolutions (give in meters per cell) result in grid maps with different sizes. Therefore, in order to evaluate the effect of the changes in map dimension and map resolution on the execution time, different map dimensions (40 × 40 m and 60 × 60 m for IRL and 60 × 60 m, 80 × 80 m for ARRL) and map resolutions (5 cm/cell and 2.5 cm/cell) are used. Both Figs. 18 and 19 illustrate that increasing the map size while keeping a fixed resolution does not show a significant increase in avg. execution time for all methods. This is because all incremental methods do not process unmapped grid cells thus simply increasing the map size does not increase the execution time. However for the same map size, increasing the resolution (i.e. decreasing the grid cell size from 5 to 2.5 cm) significantly increases the execution time. This is due to the increase in the number of mapped cells with the increase in map resolution. Comparisons on CMU-NSH, FR101 and Seattle-R environments are conducted on maps with dimensions of 40 × 40 m, 80 × 50 m and 80 × 40 m with a map resolution of 5 cm/cell. Hence, the sizes of grids used for these three environments are 800 × 800, 1600 × 1000 and 1600 × 800 respectively. Comparisons among the different methods are given next.

Since WFD method processes the entire mapped space, it degenerates to TRA method over time and requires higher avg. execution time as expected. Comparatively OFD method requires significantly low execution time for all environments and map sizes. SRFD and $WFD_{INC/IP}$ requires the lowest execution time of all methods. Avg. execution times for SRFD and $WFD_{INC/IP}$ are comparable for the IRL environment. $WFD_{INC/IP}$ method performs slightly better than SRFD method in CMU-NSH and Seattle-R environments whereas SRFD method slightly outperforms $WFD_{INC/IP}$ in ARRL and FR101 environments. It must be noted that SRFD method performs additional computation on maintaining the reachability information, hence is expected to require more execution time. Comparing the maps of the environments, it can be seen that ARRL and FR101 environments are relatively open whereas CMU-NSH and Seattle-R mainly consists of narrow corridors. This directly translates to how the frontier information is managed incrementally. With relatively long frontier contours in ARRL and FR101 environments, the frontier database employed by $WFD_{INC/IP}$ becomes inefficient with huge number of single cell updates resulting in an increase in execution time. This offsets the additional computation of SRFD for reachability maintenance. In addition the L-MX-Quadtree based frontier database employed by SRFD is able to handle contour-wise update more efficiently thus results in a better performance. However in CMU-NSH and Seattle-R environments, the frontier contours are relatively small hence does not adversely increase the execution time of $WFD_{INC/IP}$ whereas the additional reachability computation of SRFD in these environments offsets the modest improvements in small contour management, thus requires slightly higher execution time compared to $WFD_{INC/IP}$. A more fair comparison on execution times between SRFD, OFD^* and $WFD^*_{INC/IP}$ where all methods perform reachability computation reveals that SRFD is significantly efficient in generating reachable valid frontier cells.

7.2. *Evaluation of accuracy*

Efficient generation of frontier information should not come at the expense of accuracy. Therefore, we evaluated the accuracy of SRFD method with respect to the traditional method for all five environments. Accuracy of the WFD, $WFD_{INC/IP}$ and OFD with respect to TRA is also provided.

7.2.1. *Quantitative analysis of accuracy*

For each environment type, both false positive detections and false negative non-detections using SRFD, WFD, $WFD_{INC/IP}$ and OFD methods are recorded with respect to the frontier information generated using the TRA method. False positive detections are

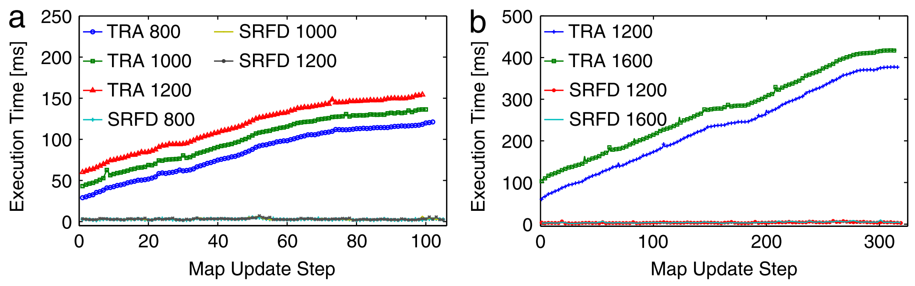
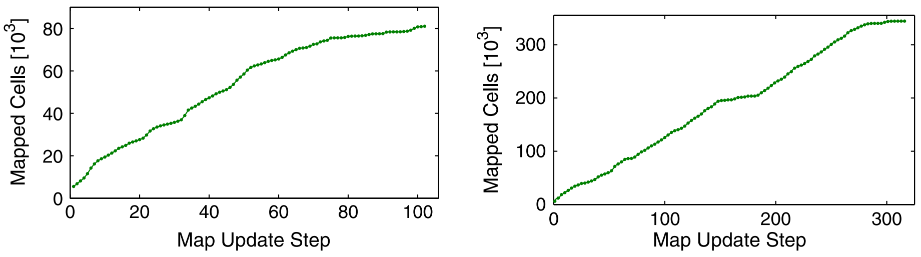


Fig. 16. Comparison of execution times for frontier generation between SRFD and TRA methods with different map dimensions for (a) IRL (b) ARRL environments.



(a) IRL environment.

(b) ARRL environment.

Fig. 17. Progression of no. of mapped cells during exploration.

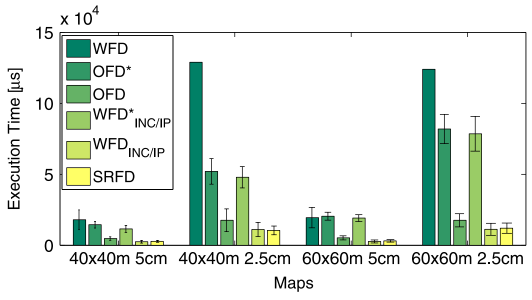


Fig. 18. Execution time comparison for IRL environment.

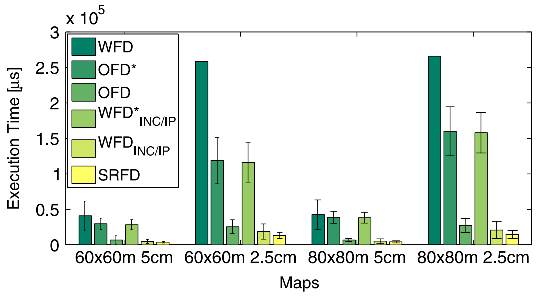


Fig. 19. Execution time comparison for ARRL environment.

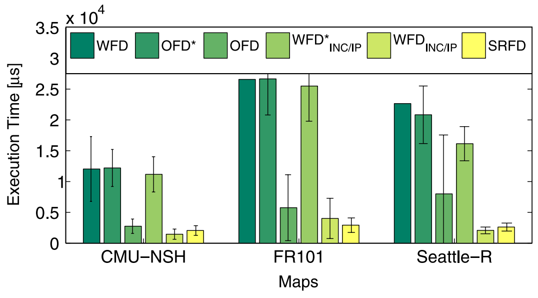


Fig. 20. Execution time comparison for CMU-NSH, FR101 and Seattle-R environments.

cells erroneously reported as frontiers whereas false negative non-detections are cells erroneously not reported as frontier cells with respect to the frontier cells reported by TRA method. This is done across all the map update steps. Comparison of the false positive and false-negative percentages for all environments are given in Figs. 21–25. It can be seen that SRFD method exhibits the lowest false-positive error percentage and is consistently close to 0%. This is due to SRFD's ability to differentiate between reachable and non-reachable areas allowing the filtering out of "phantom-frontiers". WFD, $WFD_{INC/IP}$ and OFD methods do not specifically check for reachability of the generated frontiers and thus result in generation of more false-positive frontiers. Hence they must be filtered out before being used to generate the next exploration decision. Since the reachability is not computed incrementally, the entire mapped space has to be processed at each update step. This requires additional execution time which increases with the increase in the number of mapped cells as described using Fig. 17 and reported for OFD^* and $WFD^*_{INC/IP}$ in Figs. 18–20. Hence it defeats the purpose of efficient incremental frontier generation. However, all methods indicate relatively low false-negative percentages and SRFD method's false-negative percentage is comparable to $WFD_{INC/IP}$ and WFD for most of the mission time steps and its effect on exploration missions is evaluated qualitatively.

7.2.2. *Qualitative analysis of accuracy*

The frontiers generated by the SRFD method are qualitatively compared with the ones generated by the TRA method and it is revealed that the SRFD method generates all the major frontier contours identified by the traditional approach. Figs. 26 and 27 illustrate two example qualitative comparisons of the accuracy for the SRFD method for IRL and ARRL environments respectively. The retrieved frontiers are marked in red. The minor frontier non-detection errors (i.e. false-negatives) are due to missing individual frontier cells that do not contribute to any significant degradation of the exploration missions as the information about all the major frontier contours are still available.

Two frontier based exploration missions are conducted, one with TRA and the other with SRFD as the frontier generation method in an indoor environment to validate that the frontiers generated by SRFD method are sufficient for complete exploration.

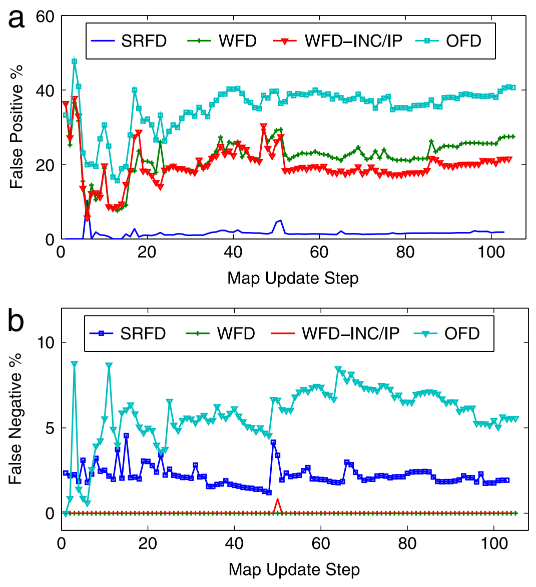


Fig. 21. Comparison of (a) false positives and (b) false negatives for IRL environment.

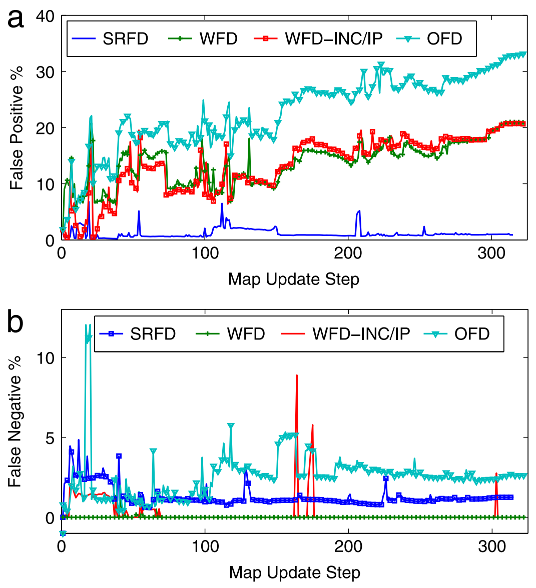


Fig. 22. Comparison of (a) false positives and (b) false negatives for ARRL environment.

Utility based on the trade-off between information gain and cost is used for target selection at each step [8]. For both experiments the maximum usable sensor range is set to 4 m during mapping and the map resolution is set to 5 cm/cell. The mission with SRFD as frontier generation method also utilized its fast frontier refreshing rate to invoke the exploration decision at a higher frequency (i.e. at each frontier refreshing step) as suggested in [16]. Fig. 28 depicts the results of the two exploration missions together with the paths traversed by the robot. Both missions had the same starting pose and generally followed the same exploratory path. The exploration mission with frontiers generated using TRA method and without frequent decision invocation required 224.16 m to complete exploration whereas the mission with SRFD method and frequent decision invocation based exploration required only 207.31 m. It can be observed that the traditional approach for exploration sometimes lead the robot to the very end of a dead-end/wall (Fig. 28(a)) whereas frequent decision making facilitated by SRFD is able to return quickly from such instances thus leading to a more efficient exploration mission, hence confirming the utility of efficient frontier generation for exploration missions.

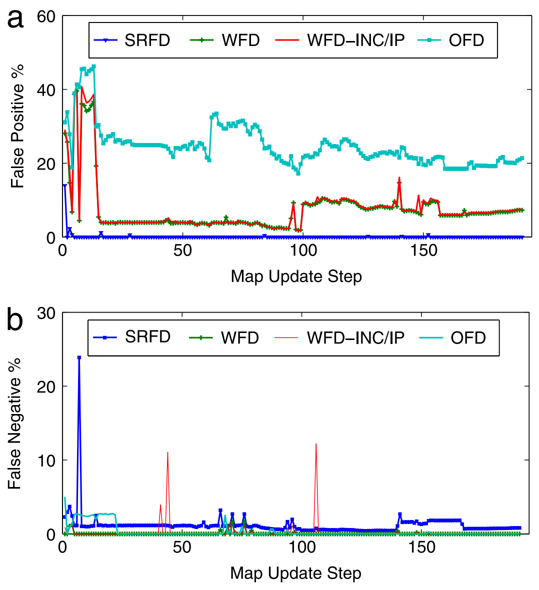


Fig. 23. Comparison of (a) false positives and (b) false negatives for CMU-NSH environment.

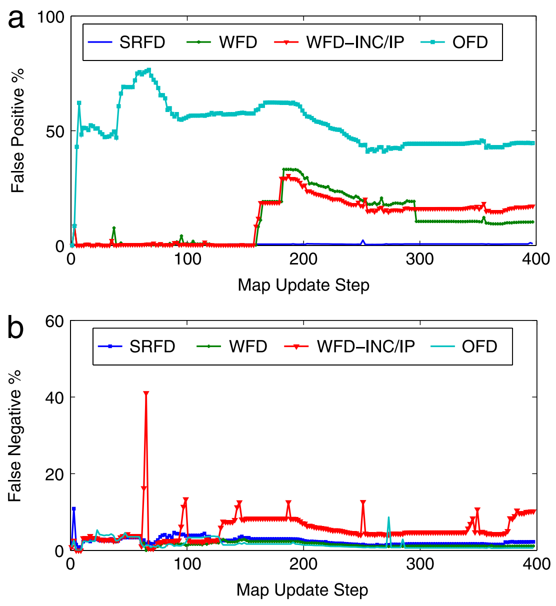


Fig. 24. Comparison of (a) false positives and (b) false negatives for FR101 environment.

7.3. *The search depth of reachability graph*

Retrieving reachability of the cells in occupancy grid maps requires searching the spanning tree proposed in Section 5.2 and it is important to minimize the number of search steps involved. Cells abstracted by visited nodes in G_k require only one search step whereas reachable non-visited cells require multiple search steps and the theoretical bounds on the depths of associated nodes are given in Section 5.2.1. However these bounds are not explicitly enforced during the update algorithm as it requires additional operations to update depths of each node in the tree. Therefore the search depths of the trees generated without the node depth

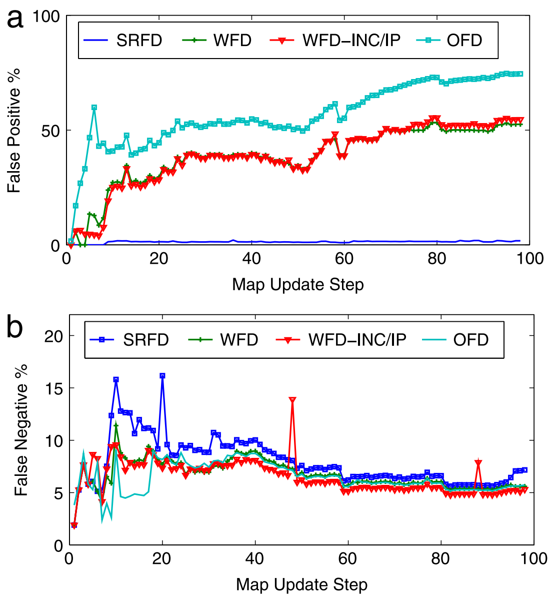
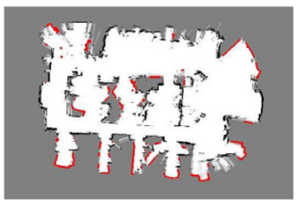
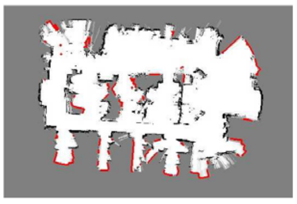


Fig. 25. Comparison of (a) false positives and (b) false negatives for Seattle-R environment.

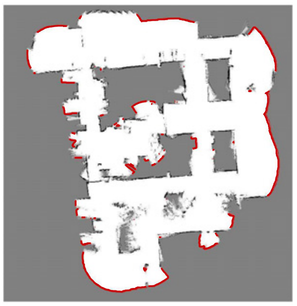


(a) 1.63% f-positive, 2.08% f-negative errors.

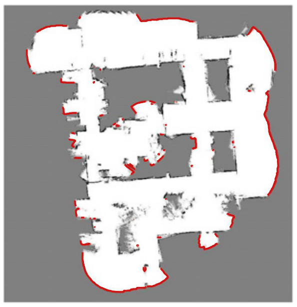


(b) Frontiers detected at update step 95 by TRA method.

Fig. 26. Qualitative comparison of frontiers generated in the IRL environment by (a) SRFD and (b) TRA methods during map update step 95. Detected frontiers are marked in red color. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



(a) 0.98% f-positive, 1.22% f-negative errors.



(b) Frontiers detected at update step 300 by TRA method.

Fig. 27. Qualitative comparison of frontiers generated in the AARL environment by (a) SRFD and (b) TRA methods during map update step 300. Detected frontiers are marked in red color. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

checks are recorded for IRL and ARRL environments to evaluate their conformance to the theoretical values. For both environments the average number of search steps and the standard deviation to check for reachability of each cell in reachable non-visited safe-patches at each map update step are recorded along a mapping mission. This is repeated with different maximum usable sensor range values of 3 m, 4 m and 5 m that provide theoretical search step bounds of 7, 9 and 11 respectively, considering $\delta d = 0.5$ m.

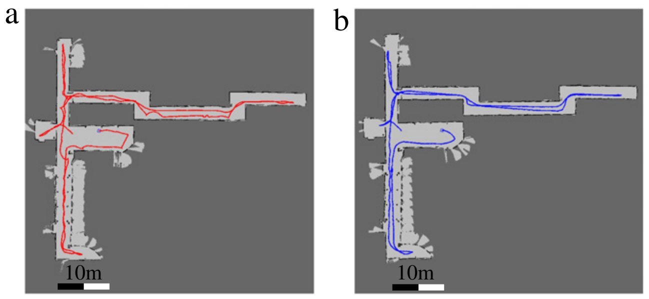


Fig. 28. Exploring an indoor environment using frontiers generated from (a) TRA and (b) SRFD method. TRA method without frequent decision invocation required 224.16 m. SRFD with frequent decision invocation required only 207.31 m.

Fig. 29(a) and (b) depict the percentage of reachable non-visited cells out of mapped cells at each map update step for the three different sensor range values for both environments. The considerable percentages of non-visited reachable cells present throughout the missions confirm the requirement to minimize the search steps used for reachability check.

It is observed in Figs. 30 and 31 that for both environments and the three different sensor range values, the average number of search steps and its deviation are within the theoretical bounds of 7, 9 and 11 respectively even without explicitly enforcing them in the update algorithms. This results in efficient reachability detection of mapped-free-space contours, hence an efficient incremental frontier detection strategy.

8. Conclusion and future work

In this paper a set of algorithms and supporting data structures for efficient frontier extraction and management is proposed. The presented work is based on incrementally processing only the modified portions of the occupancy grid map to extract information about local frontier updates and then managing a database across update steps to ensure global consistency of the detected frontiers. SRFD differs from earlier works on efficient frontier extraction strategies by integrating incremental detection of safe-space and incremental detection of reachability information in to the frontier extraction strategy itself, alleviating the need for a separate criteria to filter invalid (i.e. unsafe and unreachable) frontiers from detected frontiers. Incremental updates to frontier contours are managed using the L-MX-Quadtree data-structure which is based on MX-Quadtree and is shown to provide efficient access to both frontier contours and individual frontier cells.

Experiments conducted using real world data sets confirmed that SRFD provides a bound on the execution time for frontier information extraction operations through out entire exploration missions. This is due to processing of only locally updated map information and the use of an efficient contour database and is shown to be comparable to state of the art incremental approaches. In addition, the accurate representation of safe and reachable map information to filter out phantom frontiers results in a very high level of detection accuracy compared to the state of the art methods.

An exploration mission conducted in an indoor environment validated that the frontiers generated by SRFD method is sufficient for complete exploration. Additionally it also demonstrated that low computational footprint of SRFD could be used to invoke the exploration decision at high frequency which leads to efficient exploration. Future work includes the use of SRFD method with its high detection accuracy for the development of more efficient exploration algorithms that leverage on high frontier refreshing rates. Additionally, the single robot based map generation assumed

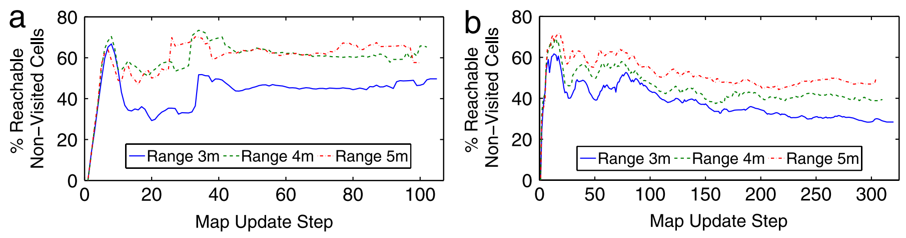
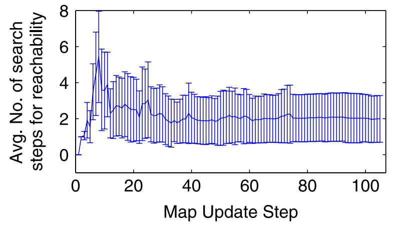
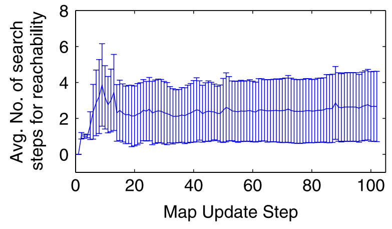


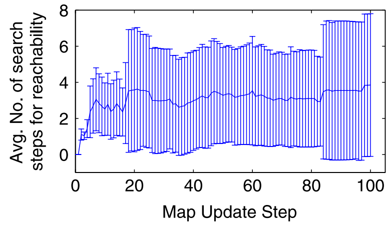
Fig. 29. Comparison of the percentage of non-visited reachable grid cells, across map update steps, in occupancy grid maps generated with 3 m, 4 m and 5 m sensor ranges for (a) IRL and (b) ARRL environment.



(a) Range 3 m.

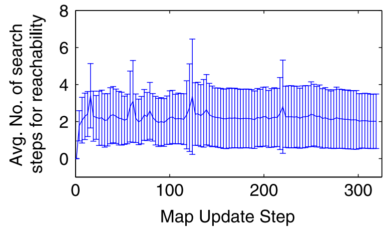


(b) Range 4 m.

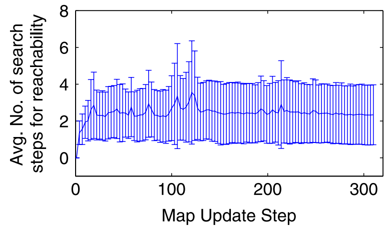


(c) Range 5 m.

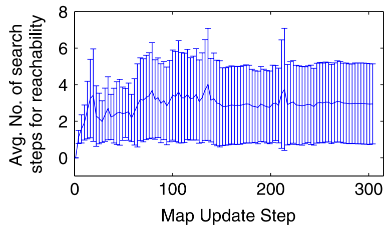
Fig. 30. Progression of the avg. no. of search steps for reachability check for IRL environment.



(a) Range 3 m.



(b) Range 4 m.



(c) Range 5 m.

Fig. 31. Progression of the avg. no. of search steps for reachability check for ARRL environment.

during incremental reachability detection can be relaxed in the future to investigate incremental reachability computation and frontier extraction from maps generated by multiple robots.

Acknowledgments

We gratefully acknowledge Nick Roy, Cyrill Stachniss and Dirk Haehnel for making sensor data sets for CMU-NSH, FR101 and Seattle-R environments publicly available. We also acknowledge the valuable suggestions conveyed by the reviewers of this article.

References

[1] D. Calisi, A. Farinelli, L. Iocchi, D. Nardi, Multi-objective exploration and search for autonomous rescue robots, J. Field Robot. 24 (8–9) (2007) 763–777.
[2] C. Weisbin, G. Rodriguez, NASA robotics research for planetary surface exploration, IEEE Robot. Autom. Mag. 7 (4) (2000) 25–34.
[3] A. Bennett, J. Leonard, A behavior-based approach to adaptive feature detection and following with autonomous underwater vehicles, IEEE J. Ocean. Eng. 25 (2) (2000) 213–226.
[4] W. Burgard, M. Moors, D. Fox, R. Simmons, S. Thrun, Collaborative multi-robot exploration, in: Proceedings IEEE International Conference on Robotics and Automation, Vol. 1, 2000, pp. 476–481.
[5] C. Tovey, S. Koenig, Improved analysis of greedy mapping, in: Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on, Vol. 4, IEEE, 2003, pp. 3251–3257.
[6] A. Elfes, Using occupancy grids for mobile robot perception and navigation, Computer 22 (6) (1989) 46–57.
[7] B. Yamauchi, A frontier-based approach for autonomous exploration, in: Proceedings., 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation, 1997, pp. 146–151.
[8] H.H. González-Baños, J.-C. Latombe, Navigation strategies for exploring indoor environments, Int. J. Robot. Res. 21 (10–11) (2002) 829–848.
[9] R. Lakaemper, L.J. Latecki, X. Sun, D. Wolter, Geometric robot mapping, in: Discrete Geometry for Computer Imagery, Springer, 2005, pp. 11–22.
[10] C. Stachniss, G. Grisetti, W. Burgard, Information gain-based exploration using Rao–Blackwellized particle filters, in: Proceedings of Robotics: Science and Systems, Cambridge, USA, 2005.
[11] M. Kulich, J. Faigl, L. Preucil, On distance utility in the exploration task, in: Robotics and Automation (ICRA), 2011 IEEE International Conference on, IEEE, 2011, pp. 4455–4460.
[12] G. Grisetti, C. Stachniss, W. Burgard, GMapping. URL: http://openslam.org/gmapping.html.
[13] S. Kohlbrecher, J. Meyer, Hector SLAM. URL: http://wiki.ros.org/hector_slam.
[14] D. Holz, N. Basilico, F. Amigoni, S. Behnke, Evaluating the efficiency of frontier-based exploration strategies, in: Proceedings of Joint 41st International Symposium on Robotics and 6th German Conference on Robotics, 2010.
[15] M. Keidar, G.A. Kaminka, Efficient frontier detection for robot exploration, Int. J. Robot. Res. 33 (2013) 215–236.
[16] F. Amigoni, A. Quattrini Li, D. Holz, Evaluating the impact of perception and decision timing on autonomous robotic exploration, in: Mobile Robots (ECMR), 2013 European Conference on, IEEE, 2013, pp. 68–73.
[17] P. Senarathne, D. Wang, Frontier based exploration with task cancellation, in: Safety, Security, and Rescue Robotics, SSRR, 2014 IEEE International Symposium on, 2014, pp. 1–6. http://dx.doi.org/10.1109/SSRR.2014.7017658.
[18] A. Visser, Xingrui-Ji, M. van Ittersum, L. González Jaime, L. Stancu, Beyond frontier exploration, in: RoboCup 2007: Robot Soccer World Cup XI, in: Lecture Notes in Computer Science, vol. 5001, Springer, Berlin, Heidelberg, 2008, pp. 113–123.
[19] R. Wein, J.P. Van Den Berg, D. Halperin, The visibility–voronoi complex and its applications, in: Proceedings of the Twenty-First Annual Symposium on Computational Geometry, ACM, 2005, pp. 63–72.
[20] M. Quigley, K. Conley, B.P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A.Y. Ng, ROS: an open-source robot operating system, in: ICRA Workshop on Open Source Software, 2009.
[21] R.C. Gonzalez, R.E. Woods, Digital Image Processing, second ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
[22] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, et al., Introduction to Algorithms, Vol. 2, MIT Press, Cambridge, 2001.
[23] C.Y. Lee, An algorithm for path connections and its applications, IRE Trans. Electron. Comput. (3) (1961) 346–365.
[24] M. Juliá, A. Gil, L. Paya, O. Reinoso, Potential field based integrated exploration for multi-robot teams, in: Proceedings of the 5th International Conference on Informatics in Control, Automation and Robotics, pp. 308–314.

[25] A. Mobarhani, S. Nazari, A.H. Tamjidi, H.D. Taghirad, Histogram based frontier exploration, in: Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on, IEEE, 2011, pp. 1128–1133.
[26] D. Puig, M. Garcia, L. Wu, A new global optimization strategy for coordinated multi-robot exploration: Development and comparative evaluation, Robot. Auton. Syst. 59 (9) (2011) 635–653. http://dx.doi.org/10.1016/j.robot.2011.05.004.
[27] P. Senarathne, D. Wang, A two-level approach for multi-robot coordinated exploration of unstructured environments, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, ACM, 2012, pp. 274–279.
[28] A. Bautin, O. Simonin, F. Charpillet, MinPos: A novel frontier allocation algorithm for multi-robot exploration, in: Intelligent Robotics and Applications, Springer, 2012, pp. 496–508.
[29] C. Stachniss, O. Mozos, W. Burgard, Speeding-up multi-robot exploration by considering semantic place information, in: Proceedings IEEE International Conference on Robotics and Automation, 2006, pp. 1692–1697. http://dx.doi.org/10.1109/ROBOT.2006.1641950.
[30] K.M. Wurm, C. Stachniss, W. Burgard, Coordinated multi-robot exploration using a segmentation of the environment, in: Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008, pp. 1160–1165.
[31] B. Lau, C. Sprunk, W. Burgard, Efficient grid-based spatial representations for robot navigation in dynamic environments, Robot. Auton. Syst. 61 (10) (2013) 1116–1130.
[32] P. Senarathne, D. Wang, Z. Wang, Q. Chen, Efficient frontier detection and management for robot exploration, in: Cyber Technology in Automation, Control and Intelligent Systems (CYBER), 2013 IEEE 3rd Annual International Conference on, IEEE, 2013, pp. 114–119.
[33] S. Gottschalk, M.C. Lin, D. Manocha, OBBTree: a hierarchical structure for rapid interference detection, in: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'96, ACM, New York, NY, USA, 1996, pp. 171–180. http://dx.doi.org/10.1145/237170.237244.
[34] N. Kalra, D. Ferguson, A. Stentz, Incremental reconstruction of generalized Voronoi diagrams on grids, Robot. Auton. Syst. 57 (2) (2009) 123–128.
[35] J. Holm, K. De Lichtenberg, M. Thorup, Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity, J. ACM 48 (4) (2001) 723–760.
[36] Z. Galil, G.F. Italiano, Data structures and algorithms for disjoint set union problems, ACM Comput. Surv. 23 (3) (1991) 319–344.
[37] M. De Berg, M. Van Kreveld, M. Overmars, O.C. Schwarzkopf, Computational Geometry, Springer, 2000.
[38] H. Samet, The quadtree and related hierarchical data structures, ACM Comput. Surv. 16 (2) (1984) 187–260.
[39] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, Commun. ACM 33 (6) (1990) 668–676. http://dx.doi.org/10.1145/78973.78977.
[40] J.L. Bentley, Solutions to klee's rectangle problems, Technical report, Carnegie-Mellon Univ., Pittsburgh, PA, 1977.
[41] A. Howard, N. Roy, The robotics data set repository (radish), 2003. URL: http://radish.sourceforge.net/.
[42] S. Thrun, W. Burgard, D. Fox, Probabilistic Robotics, MIT Press, 2005.

P.G.C.N. Senarathne received his B.Sc. degree in Computer Science and Engineering from University of Moratuwa, Sri Lanka in 2009. Currently he is a Ph.D. candidate at the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. His research interests include single and multi-robot exploration and computational geometric applications to robot exploration. He is also generally interested in developing autonomous navigation solutions for mobile robots.

Danwei Wang received his Ph.D. and M.S.E. degrees from the University of Michigan, Ann Arbor in 1989 and 1984, respectively. He received his B.E. degree from the South China University of Technology, China in 1982. He is professor in the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. He is director of EXQUISITUS, Centre for E-City and deputy director of the Robotics Research Centre, NTU. He has served as general chairman, technical chairman and has held various positions in international conferences. He has served as an associate editor of Conference Editorial Board, IEEE Control Systems Society. He is an associate editor of International Journal of Humanoid Robotics and invited guest editor of various international journals. He was a recipient of Alexander von Humboldt fellowship, Germany. He has published widely in the areas of iterative learning control, repetitive control, fault diagnosis and failure prognosis, satellite formation dynamics and control, as well as manipulator/mobile robot dynamics, path planning, and control.