Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 438576, 11 pages
http://dx.doi.org/10.1155/2013/438576

Research Article

Dynamic Detection of Topological Information from Grid-Based Generalized Voronoi Diagrams

Long Qin, Quanjun Yin, Yabing Zha, and Yong Peng

College of Information System and Management, National University of Defense Technology, Hunan, Changsha 410073, China

Correspondence should be addressed to Quanjun Yin; yin_quanjun@163.com

Received 11 July 2013; Revised 3 October 2013; Accepted 3 October 2013

Academic Editor: Piermarco Cannarsa

In the context of robotics, the grid-based Generalized Voronoi Diagrams (GVDs) are widely used by mobile robots to represent their surrounding area. Current approaches for incrementally constructing GVDs mainly focus on providing metric skeletons of underlying grids, while the connectivity among GVD vertices and edges remains implicit, which makes high-level spatial reasoning tasks impractical. In this paper, we present an algorithm named Dynamic Topology Detector (DTD) for extracting a GVD with topological information from a grid map. Beyond the construction and reconstruction of a GVD on grids, DTD further extracts connectivity among the GVD edges and vertices. DTD also provides efficient repair mechanism to treat with local changes, making it work well in dynamic environments. Simulation tests in representative scenarios demonstrate that (1) compared with the static algorithms, DTD generally makes an order of magnitude improvement regarding computation times when working in dynamic environments; (2) with negligible extra computation, DTD detects topologies not computed by existing incremental algorithms. We also demonstrate the usefulness of the resulting topological information for high-level path planning tasks.

1. Introduction

In robotics, it is essential for a mobile robot to be able to construct and maintain consistent models of its working space. With such an internal description of the environment, most spatial reasoning tasks, such as path planning, self localization, and collision detection, are feasible. Generally speaking, the efficiency of a spatial reasoning algorithm depends on the size and representation of the underlying space model. Therefore, constructing a sparse, adequate, and well-organized representation of the environment is a key issue in the successful design of a mobile robot.

Common representations for describing the environment include (but are not limited to) uniform [1] and nonuniform grid maps [2], probabilistic roadmaps [3], waypoint graph [4], and Generalized Voronoi Diagrams (GVDs). We first give a definition of the GVDs. Let S denote a set of n sites (e.g. points, curves, line segments, and polygons) in a plane D. For each site $p \in S$, the GVD region of p is defined as

$$\operatorname{reg}(p) = \{c \mid c \in D \text{ and } \operatorname{dis}(c,p) \le \operatorname{dis}(c,q) \quad \forall q \in S - \{p\}\}, \tag{1}$$

referring to a set of points that keep p as the nearest site than the others. The boundary that divides two regions is named as a GVD edge which can be denoted as

$$\operatorname{edge}(p,q) = \{c \mid c \in \operatorname{reg}(p) \text{ and } c \in \operatorname{reg}(p)\}. \tag{2}$$

Due to the prevalence of grid-based environment representations in robotics, GVDs built on grids are widely used and outperform the other representations in extracting sparse but adequate environment skeletons [6, 7]. The advantages of employing such a skeleton are twofold. Firstly, it can serve as a roadmap that significantly reduces the complexity of search problems. Secondly, it provides maximum clearance to the sites which are usually considered as obstacles. Due to these advantages, research on how to construct the GVDs efficiently has drawn significant attention in recent years.

Several algorithms for computing grid-based GVDs have been studied, for example, the Burshfire algorithm [8] and its improved versions (Dynamic Brushfire [7] and a novel approach proposed by Lau et al. in [6]). However, the Brushfire algorithm failed to update local changes efficiently when there are only partial areas needing repair; instead it just abandons the existing GVD and builds a new one

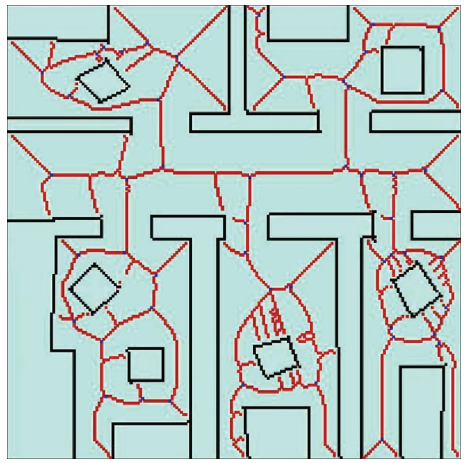


Figure 1: Those points which are equidistant from at least three sites are denoted as GVD vertices. As a consequence, a plane can be represented as a partition and thus is called the GVD of S. As an example, Figure 1 represents the GVD of an indoor environment which first appeared in [5] and we build its GVD here by our proposed algorithm, Dynamic Topology Detector (DTD). The GVD of an indoor environment; red lines denote the GVD edges, blue dots are GVD vertices, and black polygons represents walls and furniture, that is, the GVD sites.

from scratch. Such an inefficient strategy is unacceptable in a dynamic environment. To deal with this problem, novel algorithms which possess local update mechanism are then proposed in [6, 7]. These algorithms can incrementally construct and reconstruct GVDs on grids, providing metric results as shown in Figure 2.

As Figure 2 shows, the metric results commonly maintain three matrices, $obst_s$, $dist_s$, and $voro_s$, to represent a GVD. The matrix $dist_s$ keeps discrete or actual Euclidean distance between an arbitrary entry (denoted by s) and the site cell from which s propagates; the matrix $obst_s$ registers the site identifier and the coordinate of the exact site cell to which s is currently the closest; the matrix $voro_s$ is a Boolean matrix which indicates whether s is a GVD cell. Compared to occupancy grids, these matrices provide algorithms, such as A^* and D^*, with more heuristic information and reduced search space. Therefore, reasoning in GVD matrices is efficient in terms of speed and the quality of the paths produced. For example, when using A^* to find a path between two cells, the whole search process consists of three steps. Firstly, construct the path from the start cell to its closest entry on the matrix $voro_s$ (the access cell). This can be done by iteratively exploring an adjacent cell possessing the highest value of $dist_s$ as the next move until a $voro_s$ entry is reached. Secondly, search in the $voro_s$ till the entry closest to the goal is located (the departure cell). Thirdly, construct the path from the departure cell to the goal. Since the search space is limited and all the path cells possess maximal clearance to the sites, the searching can be significantly fast and reasonable.

Although these GVD matrices provide reduced metric maps, they still suffer from memory complexity and lack of flexible access to the environment topologies. For instance, it is hard to decide topological relations between two GVD vertices or edges by only referring to the matrices mentioned before. However, it is shown to be more efficient if the ultimate metric decisions (e.g., how or exactly where to move) are delayed or localized until a high level plan based on the environment topologies has been achieved [9]. Unfortunately, the existing algorithms that are mentioned above do not provide methods for detecting the required environment topologies. This shortage hinders the application of high level reasoning algorithms. Therefore, the ability to build topological maps upon grid-based GVDs is crucial for a mobile robot that moves in a large area.

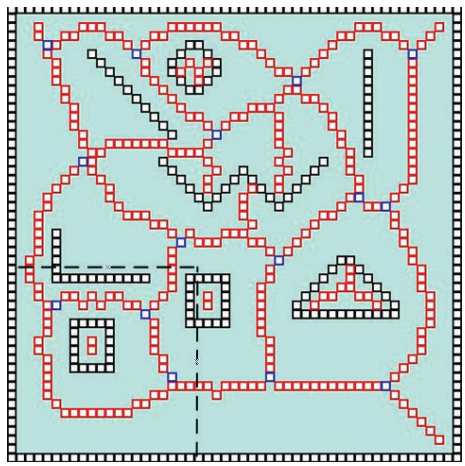
To solve this problem, we present the algorithm Dynamic Topology Detector (DTD) for extracting a GVD with topological information from a grid map. In this paper, a combined data structure is first defined to provide well-organized access employed by DTD to register the connectivity among the GVD components. The DTD is then presented both intuitively and through pseudocode in which the four execution steps (i.e., marking or freeing sites, updating GVD, thinning GVD edges, and updating GVD vertices) are discussed in order. It is proved that DTD also possesses a local repair mechanism to update the precomputed topologies when the underlying GVD is reconstructed. We compared our algorithm to current ones on several simulation scenarios and demonstrated the usefulness of the resulting topologies for high-level path planning tasks.

The outline of this paper is as follows. Section 2 discusses related techniques for GVD construction, Section 3 defines the data structure employed by DTD, Section 4 gives details about the proposed DTD algorithm, and Section 5 compares DTD to other algorithms and tests the usefulness of the resulting topological information for high-level path planning tasks. This paper ends with conclusions in Section 6.

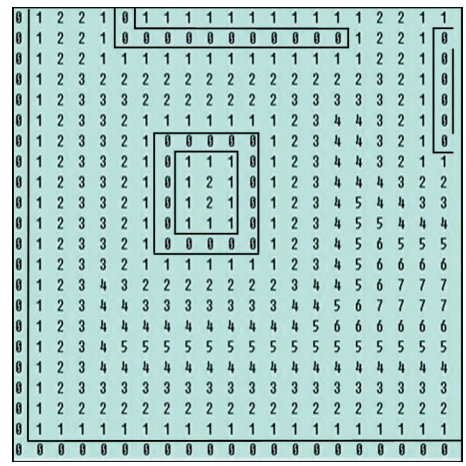
2. Related Work

In the context of robotics, GVD is a popular spatial representation for navigation and motion planning tasks. Spatial reasoning algorithms commonly take GVDs as the reduced searching space and as optimal roadmaps to save considerable computation time [10–14].

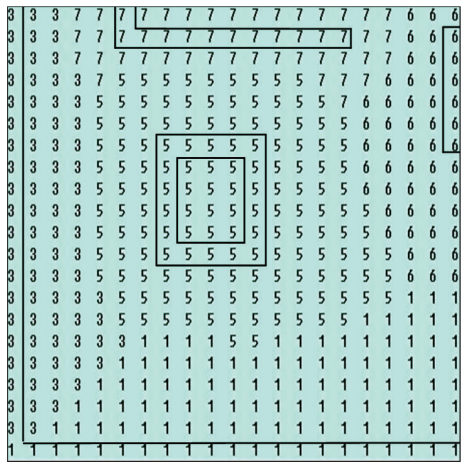
Existing algorithms for computing the GVD can be roughly divided into two kinds which operate on continuous and discrete space, respectively [15]. GVDs upon continuous space are built as a set of parametric lines or curves which separate different sites [16, 17]. There are also local update mechanisms for moving sites [18] or sites that have been inserted or deleted [19]. Such analytic methods, despite giving more accurate and sparser representation, are not practical for robots whose surrounding area is preferably modeled as grid maps. Moreover, discretizing the continuous GVD to fit the grid map is not possible because (1) different GVD edges within the same grid cell will be mixed and (2) if the exact edge coincididently lie between two grids, the discretization will either choose both the two grids to form a thick edge or simply return a detection false. Based on the above reasons, we focus on GVDs which are computed in discrete space, that is, on grids.



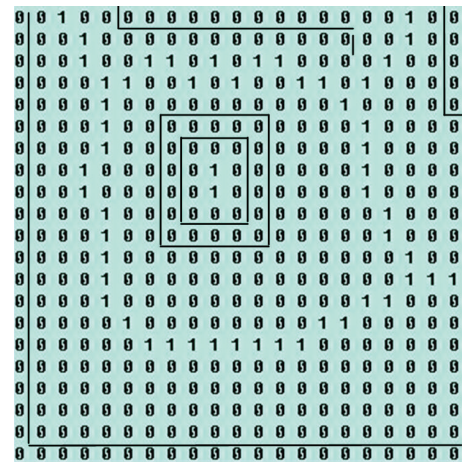
(a) Graphic representation of the sample grid-based voronoi map with size 5151 (generated by DTD)



(b) Corresponding $dist_s$ matrix where each entry keeps the integral distance to its nearest site cell



(c) Corresponding $obst_s$ matrix where each entry keeps the site identifier and exact coordinate of its nearest site cell. Here only the site identifier is explicitly represented



(d) Corresponding $voro_s$ matrix where each entry shows whether the site cell belongs to the GVD (register as 1) or not (register as 0)

Figure 2: Resulting GVD matrices constructed by the novel algorithms possessing local update mechanism: (a) is the resulting GVD, (b), (c), and (d) are metric matrices representing the left bottom 1/4 part.

In the construction of discrete GVDs, some researchers prefer fast computation using graphics hardware [20, 21]. However, this is infeasible for situations including: (1) robots with limited hardware load in real world scenarios and (2) computer generated agents performing spatial reasoning tasks in virtual reality. Therefore, much attention concentrates on hardware-independent methods. Some of the recent approaches to rebuild GVDs on grids are based on the well-known Brushfire algorithm [8]. Brushfire is based on D^* for pathfinding, possessing a priority queue *open* of the cells to propagate the change. The priority of a cell (denoted by s) is determined by its newly updated distance in $dist_s$ and cells are popped with increasing priorities. Sequentially, new cells which are adjacent to the popped ones are tested, among which newly updated cells are inserted into the *open* queue so the propagation continues. Intuitively, Brushfire propagates changes (e.g. insertion or deletion of sites) through a wavefront as shown in Figure 3 [6]. This wavefront updates GVD matrices from the source of the change and terminates when the change does not affect any more cells.

Kalra et al. in their foundational work proposed a dynamic brushfire algorithm [7] to incrementally rebuild GVDs on grids. In this algorithm, the entry value of $dist_s$ is estimated by grid steps accumulated throughout the propagation. Such an approximation potentially leads either a collision risk or overly conservative movements. To solve this problem, Scherer et al. propagate actual Euclidean distance from the exact source cell so that relative error can be significantly reduced [22]. Following this improvement, Lau et al. in their work proposed novel methods to rebuild GVDs with less computation time and fewer cell visits [6]; different from Kalra's work, their approach does not rely on site identifiers

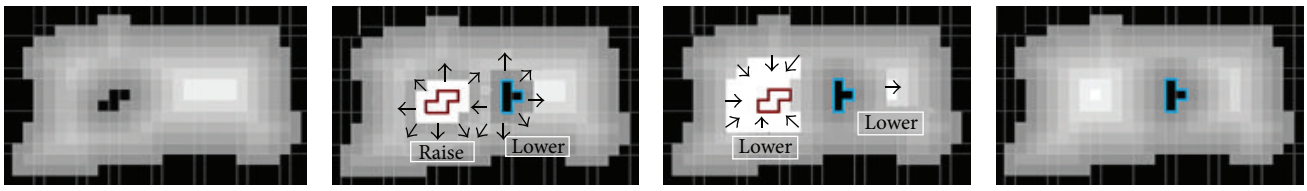


Figure 3: Distance map update between two configurations (a) and (d). Black represents occupied cells, brightness increases with distance. The inserted site (blue) initiates a "lower" wavefront shown in the intermediate steps (b) and (c) that updates the distances in the cells up to the point where a different obstacle is closer. The removed site (red outline) starts a "raise" wavefront (b) to clear the cells which lost their closest obstacle. When it comes to a halt it initiates a "lower" wavefront (c) that recomputes the distances for the cleared cells (white) on the basis of the remaining sites. This figure is cited from [6].

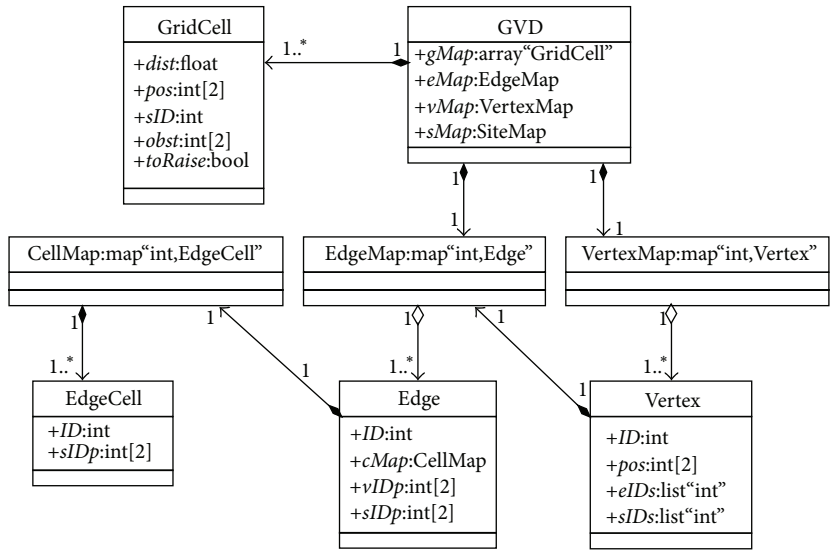


Figure 4: Class diagram for the data structure employed by DTD.

to detect GVD edges, so edges in the interior of a concave site can be also detected. Furthermore, Boris Lau et al. provided additional thinning steps using "thinning patterns" proposed by Zhang and Suen [23] to get one-cell wide edges. Therefore, the resulting edges are preferable in sparseness.

Although these algorithms are fast and efficient, they provide no mechanism for further extracting environment topologies. Details as to connectivity among GVD components (e.g. vertices and edges) remain implicit. Portugal and Rocha in their most recent work gave a solution to detect topological information for mobile robots provided with grid maps [24]. However, the extracted graph-like map provides no local repair mechanism, making it harder to fit well in dynamic environment. These information and functional shortages prevent the applications of high-level spatial reasoning methods, which are particularly unpractical for large maps.

3. Data Structure Employed by DTD

To meet the requirements for efficiently storing, repairing, and querying GVD topologies, a combined data structure which can explicitly represents spatial connectivity among GVD components [25] is employed. For each component, a corresponding container is defined to store its instances.

As Figure 4 shows, we employ hash tables to store the components and their connectivity that are detected during the construction and reconstruction of GVD. Each component instance inserted into a table is assigned with a unique identifier. For example, sites are identified by the sequence it is inserted, *vertices* are identified by their coordinates in the grid map, and edges are identified by the ID pairs indicating the two sites it divides. The semantics of related data objects and their attributes which will be quoted by DTD are listed in Table 1.

Throughout the execution of DTD, newly created instances (e.g., instances of EdgeCell, Edge, and Vertex) caused by local changes are incrementally inserted into corresponding containers (i.e., *gMap*, *eMap*, and *vMap*). At the same time, connectivity among these instances is registered in their attributes (e.g., *eIDs* within a vertex to store identifiers of its connective edges). The resulting hash tables can thus provide robots with efficient retrieval interfaces to the constructed topologies.

Table 1: The semantics table of relating data structure employed by DTD.

Classes	Attributes	Semantics
GridCell	*dist*: float	The Euclidean distance to the nearest site cell
	pos: int[2]	Coordinate of the grid cell in the grid map
	voro: bool	A mark indicating if the grid cell belongs to the GVD
	obst: int[2]	Coordinate of the nearest site cell
	sID: int	Identifier of nearest site, decided by the sequence the site is created
	toRaise: bool	A mark indicating the propagation type of this grid (raise or lower)
EdgeCell	*sIDp*: int[2]	A pair of site identifiers indicating the sites divided by this edge cell
Edge	*cMap*: CellMap	A hash table storing the edge cells indexed by their coordinates
	vIDp: int[2]	A pair of identifiers indicating the two vertices of the edge
	sIDp: int[2]	A pair of site identifiers indicating the sites divided by this edge
Vertex	*pos*: int[2]	Coordinate of the vertex in the grid map
	eIDs: list⟨int⟩	A list storing the IDs of the edges that are connective to the vertex
	sIDs: list⟨int⟩	A list storing the IDs of the sites that are connective to the vertex
GVD	*gMap*: array⟨GridCell⟩	A unique 2D array managing GVD matrices
	eMap: EdgeMap	A unique hash table storing the instances of GVD edges
	vMap: VertexMap	A unique hash table storing the instances of GVD vertices

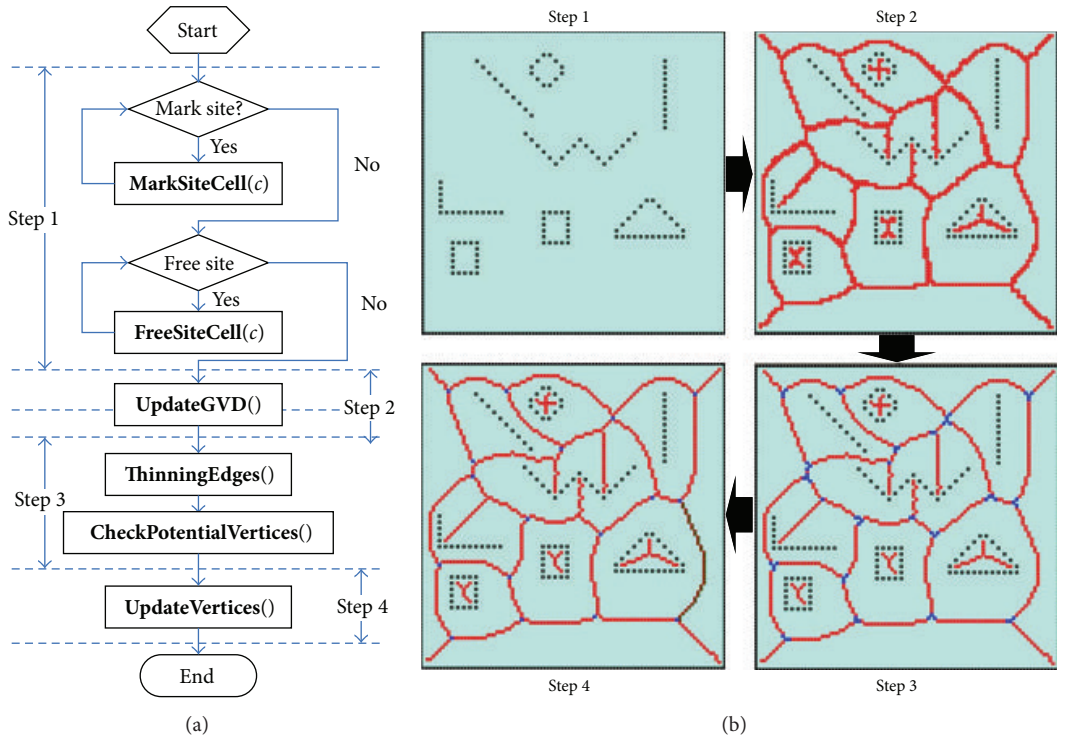


Figure 5: (a) The flowchart describing the process of rebuilding a GVD; (b) the transitions of the GVD from step 1 to step 4.

4. The DTD Algorithm

Figure 5 shows the flowchart describing the main steps of DTD. The update is started by events that cause certain cells in the grid map to transfer their state from free to occupied or vice versa, such as movement, insertion, or deletion of sites. In the first step, by repeatedly calling the function **MarkSiteCell**(*c*) and (or) **FreeSiteCell**(*c*), all changed grid cells are inserted into a priority queue *open* which is sorted by the value of *dist*. In step 2, function **UpdateGVD**() propagates the changes until there is no more affected cells remaining in *open* list. In step 3, function **ThinningEdges**() first thins the rough result to get one-cell wide GVD edges, then potential vertices are detected by function **CheckPotentialVertices**(). In step 4, function **UpdateVertices**() refines the potential vertices to get the exact vertex map. The repaired topological

```
MarkSiteCell(c, sID)
(1) obst_c ← pos_c
(2) dist_c ← 0, sID_c ← sID
(3) Insert(open, c, 0)
FreeSiteCell(c)
(4) ClearCell(c)
(5) toRaise_c ← true
(6) Insert(open, c, 0)
UpdateDVG()
(7) while open ≠ ∅
(8) c ← pop(open)
(9) if toRaise_c = true
(10) PorcessRaise(c)
(11) else
(12) if IsOcc(obst_c)
(13) ECellErase(c)
(14) ProcessLower(c)
PorcessRaise(c)
(15) for all n ∈ Adj8(c)
(16) if (obst_n ≠ ∅
          ∧¬ toRaise_n)
(17) if ¬ IsOcc(obst_n)
(18) clearCell(n)
(19) toRaise_n ← true
(20) Insert(open, n, dist_n)
(21) toRaise_c ← false
ProcessLower(c)
(22) for all n ∈ Adj8(c)
(23) if ¬ toRaise_n
(24) d ← ‖obst_c − n‖
(25) if d < dist_n
(26) dist_n ← d
(27) obst_n ← obst_c
(28) Insert(open, n, d)
(29) else MarkEdge(c, n)
```

Algorithm 1: Pseudocode for updating GVDs.

relations are incrementally registered into the data structure throughout the process. Figure 5(b) shows the transitions of the GVD in order from step 1 to step 4.

4.1. Updating the GVDs

4.1.1. Initializing and Updating the GVD. As Algorithm 1 shown is the pseudocode for initializing and updating a GVD. The initial values of the GVD matrices are set as *obst* = null, *dist* = ∞, *voro* = false, and *toRaise* = false. This is based on the fact that there is no site within the working space, and there are no sites within finite distance. When a grid cell *c* is marked as a site cell by calling function **MarkSiteCell**(*c*, *sID*), it has the $dist_c = 0$ and refer to itself as the closest site cell; that is, $obst_c = pos_c$ (lines 1-2). Conversely, when *c* is freed by calling function **FreeSiteCell**(*c*), the function **ClearCell**(*c*) resets it to the initial values (line 4). The function **Insert**(*open*, *c*, *d*) inserts *c* into *open* with priority *d*, or updates the priority if *c* is already in *open*.

In the second step, the function **UpdateGVD**() pops the next unprocessed cell *c* in order with the lowest $dist_c$ until

```
ECellErase(c)
(30) Edge e ← Find(pos_c, eMap)
(31) if e ≠ ∅
(32) Erase(c, cMap_e), voro_c ← false
(33) if cMap_e = ∅
(34) v ← Find(vID_e[0], vMap)
(35) VCellErase(v)
(36) v ← Find(vID_e[1], vMap)
(37) VCellErase(v)
(38) Erase(ID_e, eMap)
VCellErase(v)
(39) for all e ← FindEdges(eIDs_v, eMap)
(40) if vIDp_e[0] = ID_v
(41) vIDp_e[0] ← ∅
(42) if vIDp_e[1] = ID_v
(43) vIDp_e[1] ← ∅
(44) Erase(vIDp_e[0], vMap)
(45) Erase(vIDp_e[1], vMap)
```

Algorithm 2: Pseudocode for erasing cells from GVD topologies.

the queue is empty (lines 7-8). If *c* is cleared and has not yet propagated a raise wavefront, the function **PorcessRaise**(*c*) is called (lines 9-10). However, if *c* has a valid closest site cell, the function **ProcessLower**(*c*) is called. Therefore, a lower wavefront is propagated (lines 12–14). Function **ECellErase**(*c*) (line 13) erases *c* as well as its relating records from the topological data structure before function **ProcessLower**(*c*) is executed. The pseudocode for function **ECellErase**(*c*) is shown in Algorithm 2.

In **ECellErase**(*c*), the GVD edge *e* involving *c* is located (line 30) by searching *eMap* with index pos_c. Then *e* removes *c* from its cell list $cMap_e$ (line 32). If $cMap_e$ becomes empty, then the relating GVD vertices of *e* and itself become invalid and are erased from *vMap* and *eMap*, respectively (lines 33–38). In particular, function **VCellErase**(*v*) is adopted in line 35 and line 37 to reset relating records that were registered in other edge instances (lines 39–43) before the vertices are removed from *vMap* (lines 44-45).

4.1.2. Propagating the Wavefronts. All cells stored in *open* will be processed by either **PorcessLower**(*s*) or **ProcessRaise**(*s*). At the beginning, newly occupied cells call function **PorcessLower**(*s*) to launch a "lower" wavefront which propagates the changes of *dist* and *obst* to the affected cells which might be 8-connected or 4-connected grids (lines 22–29). Simultaneously, newly freed cells call function **ProcessRaise**(*s*) to launch a "raise" wavefront which clear the data of all cells whose closest site cell was the freed one (lines 15–21). During the interwoven of these two wavefronts, neighbors affected by the processed cell are again stored in *open* (line 20–28) and therefore the propagation continues.

4.1.3. Extracting the Rough GVD Edges. The rough GVD edge cells are marked and inserted into *eMap* by calling function **MarkEdge**(*s*,*n*) (line 29) when the condition $d < dist_n$ in

MarkEdge(c, n)
(46) **if** ($dist_c > 1 \vee dist_n > 1$) $\wedge$ ($obst_n \neq \emptyset$)
$\wedge$ ($obst_n \neq obst_c$) $\wedge$ ($obst_c \notin$ Adj8($obst_n$))
(47) **if** $\|pos_c - obst_n\| \leq \|pos_n - obst_c\|$
(48) *EdgeCell* s, $pos_s \leftarrow pos_c$
(49) $sIDp_s[0] \leftarrow \max(sID_n, sID_s)$
(50) $sIDp_s[1] \leftarrow \min(sID_n, sID_s)$
(51) ECellInsert(s, $dist_c$), $vror_c \leftarrow$ true
(52) **if** $\|pos_s - obst_n\| \geq \|pos_n - obst_s\|$
(53) *EdgeCell* s, $pos_s \leftarrow pos_n$
(54) $sIDp_s[0] \leftarrow \max(sID_n, sID_s)$
(55) $sIDp_s[1] \leftarrow \min(sID_n, sID_s)$
(56) ECellInsert(s, $dist_n$), $vror_n \leftarrow$ true
ECellInsert(s, *pri*)
(57) Edge $e \leftarrow$ Find($sIDp_s$, *eMap*)
(58) **if** $e = \emptyset$
(59) New(e, $sIDp_s$)
(60) $sIDp_e[0] \leftarrow sIDp_s[0]$
(61) $sIDp_e[1] \leftarrow sIDp_s[1]$
(62) Insert(e, *eMap*)
(63) Insert(s, e)
(64) Insert(s, *roughEQueue*, *pri*)

Algorithm 3: Pseudocode for marking GVD edges.

line 25 is not satisfied. As Algorithm 3 shows, we employ *Conditional-based* approach [6] to mark edges.

MarkEdge(s, n) first tests in line 46 whether at least one of c and n is not adjacent to its closest site. If n has a valid closet site that is different from and not adjacent to the closest site of c, both n and c can be the edge cell candidates. If any one of the candidate possesses the smaller distance increase when switching from its own referenced site cell to the one of the competing neighbor (tested by line 47 and 52), a new edge cell instance s is built and initiated, respectively (lines 48–50, lines 53–55).

The original approach only marks the edge cell by setting $voro_c$ = true in line 51 and 56. Here an additional function, **ECellInsert**(s), is added in the same line to insert the newly constructed edge cell s into its corresponding edge instance e in *eMap*. If such an edge is not yet existent, a new edge record is first built, initialized, and inserted into *eMap* (lines 58–62). In particular, the pair of relating site identifiers is also passed on to the newly built edge instance (lines 60-61). Finally, the edge cell s is inserted into the edge instance e and a priority queue *roughtEQueue*. *roughtEQueue* will be used by the next step to test if s is indispensable for providing connectivity in a sparse GVD edge (lines 63-64).

4.2. Thinning the Edges and Detecting Potential Vertices. In the step of Thinning the rough edges, the thinning patterns (as shown in Figure 6) proposed by Lau et al. [26] are first employed to erode two-cell-wide edges. The input taken by this thinning is the priority queue, *roughtEQueue*, which involves all edge cells that are newly created by **MarkEdge**(s, n). All the cells in *roughtEQueue* are processed in two phases. In phase 1, by modifying edge cells that are enclosed by 4-connected edges (Such cells are detected by matching pattern P8_3) as unoccupied, erroneously connected edges can be separated. In phase 2, cells are popped from the priority queue in increasing order of distance. If a popped cell has more than one neighbor edge cell and none of the patterns shown in Figure 6 match its location, then it is redundant and can be removed from *eMap* as well as *roughtEQueue* without destroying the connectivity.

CheckPotentialVertex()
(65) **for** each $s \in$ *roughtEQueue*
(66) Vertex v, $pos_v \leftarrow pos_s$
(67) Insert($sIDs_v$, $sIDp_s$)
(68) Insert($eIDs_v$, BuildEdgeID($sIDp_s$))
(69) *count* $\leftarrow 1$
(70) **for** each $n \in$ AdjECell(s)
(71) **if** $sIDp_s \neq sIDp_n$
(72) Insert($sIDs_v$, $sIDp_n$)
(73) Insert($eIDs_v$, BuildEdgeID($sIDp_n$))
(74) *count* $\leftarrow$ *count* + 1
(75) **if** count > 2
(76) Insert(v, *vertexPQueue*, *count*)
(77) **else** delete v

Algorithm 4: Pseudocode for detecting potential GVD vertices.

After the thinning is done, it is significant that, some of the edge cells are connected to the neighbor cells which belong to different edges (i.e., junctions). These cells are potential GVD vertices. Therefore, we detect these cells and enqueue them into *vertexPQueue*, a priority queue which orders the cells by increasing the number of adjacent edge cells. The detection of these potential *vertices* is accomplished by calling function **CheckPotentialVertices**(); its pseudocode is as shown in Algorithm 4.

As shown in Algorithm 4, for each edge cell s that is newly inserted into *eMap*, an instance of vertex v is first built at its location (lines 65-66). The site pair of s is inserted into the site list of v (line 67); the identifier of the edge which s belongs to is determined by $sIDp_s$ and inserted into the relating edge list of v (line 68). Then the function tests each edge cell n that is adjacent to s to check if n belongs to a different edge from the one which s belongs to (line 71). If it is true, v will merge the site pair of n into its site list (line 72). Any vertex instance v that contains more than one site pairs will be inserted into the priority queue *vertexPQueue* (line 76), or it will be deleted (line 77).

4.3. Updating the Vertices. In the third step, we use the function **CheckPotentialVertex**() to detect potential GVD vertices from newly updated edge cells. The idea is that a vertex is selected because there is more than one adjacent cell that belongs to a different edge. Relying on such a loose condition, we roughly identified some redundant cells surrounding the exact junctions (as shown in Figure 7(a)). In order to refine these cells and finally locate the exact GVD vertices, an additional function, **UpdateVertices**(), is used to fulfill the fourth step of the detection.

The pseudocode of function **UpdateVertices**() is as shown in Algorithm 5. In this algorithm, all the potential

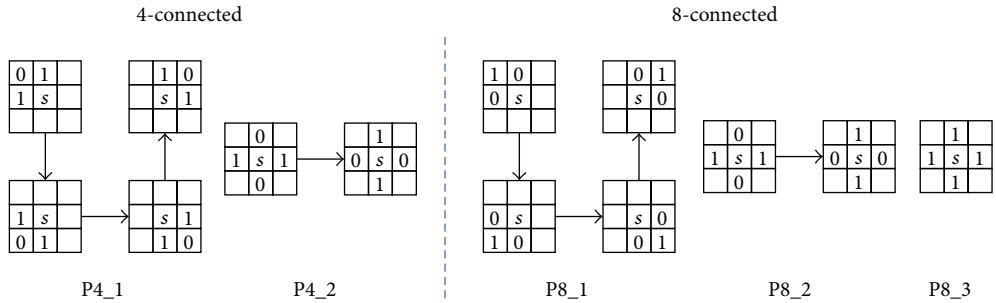


Figure 6: Patterns used by edge thinning. Arrows indicate application of rotated copies.

vertices are stored in a priority queue *vertexPQueue*. A vertex's priority is the count of adjacent edge cells whose dependent GVD edges are mutually different. The bigger the count is, the higher the priority will be. We first test each potential vertex to know whether there are at least three edge cells within its 8-inter grids. All the vertices which do not satisfy such a condition are erased from the queue because a vertex, according to the definition of GVD, must be the junction of at least three GVD edges.

Despite significant reduction of the entries (as the intermediate state shown in Figure 7(b)), there are still redundant vertices in the priority queue. However, with the highest priority, the popped vertex possesses the most connectivity than its neighbors, so it will be preserved as an exact vertex while its neighbors are erased. Before v is inserted into *vMap* (line 85), all its relating edges will update their vertex identifier pair so as to accord with the changes (lines 82–84). Such a process goes on till vertexPQueue is empty.

UpdateVertices()
(78) **while** *vertexPQueue* $\neq \emptyset$
(79) Vertex $v \leftarrow$ pop(*vertexPQueue*)
(80) **for** all $n \in$ AdjVCell(v)
(81) Erase(n, *vertexPQueue*)
(82) **for** all $e \leftarrow$ FindEdges($eIDs_v$, *eMap*)
(83) **if** $vIDp_e[0] = \emptyset$: $vIDp_e[0] \leftarrow ID_v$
(84) **else** $vIDp_e[1] \leftarrow ID_v$
(85) Insert(v, *vMap*)

Algorithm 5: Pseudocode for updating GVD vertices.

5. Experiments and Analysis

In this section, we employed statistical methods to compare our algorithm with other competing methods on some simulated scenarios. We also demonstrated the usefulness of the resulting GVD topologies detected by DTD to high-level path planning tasks.

5.1. Comparison to Other Algorithms. We compared our algorithm to Brushfire, Dynamic Brushfire, and method proposed by Boris Lau et al. (we abbreviate it as BL below) discussed in Section 2 on four scenarios as shown in Figure 8.

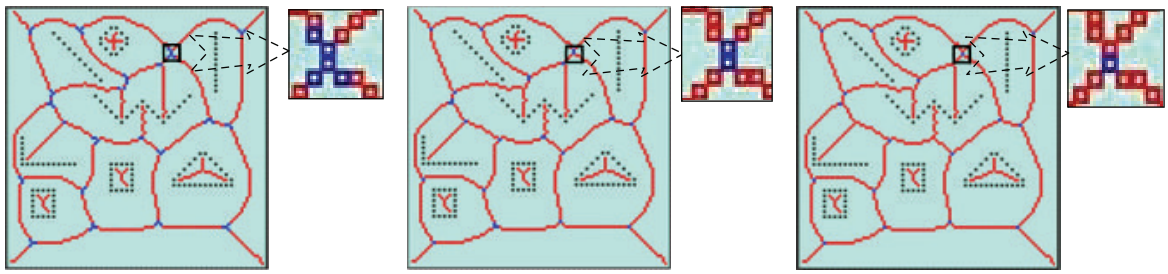
Among these methods, the Brushfire is a static method while the others are capable of local reconstruction. The first scenario was an static environment in which all the sites are predefined and fixed. The environment we used was a 201×201 grid map which had approximate 20% cells occupied by several predefined sites (as shown in Figure 8(a)). The remaining three scenarios (Figures 8(b), 8(c), and 8(d)) allowed a part of these sites (75%, 50%, and 25%, resp.) to change their position and (or) shape randomly every 5 seconds. Such kind of dynamic environment occurs frequently in robotics which therefore needs an efficient repair mechanism. A centralized planner with a global sensor was simulated and required to rebuild the GVD in every 10 seconds. We ran each algorithm on each scenario for 100 times. The tests were all done by C++ implementation of the algorithms, running on an Intel Xeon Processer.

The comparison of the performances among the four approaches are shown in Table 2 and Figure 9 (in execution time) and Table 3 and Figure 10 (in cell visits). From the relating tables and figures we can see that, for the first scenario (i.e., the global construction with no prior computation), the extra operations which enable local repair make Dynamic Brushfire, BL, and our algorithm slightly slower than Brushfire. This results accords with conclusions made by Kalar et al. [7].

For the other three scenarios, along with the decreasing rate of changing sites, DTD performs better and better than Brushfire. Although additional topological information is computed, DTD still outperforms Dynamic Brushfire due to fewer cell visits. As for BL, the extra computation only increased the time by 3.4% in the worst case. However, such a computation increment is acceptable and gains environment topologies which were not explicitly provided by BL and other competing methods.

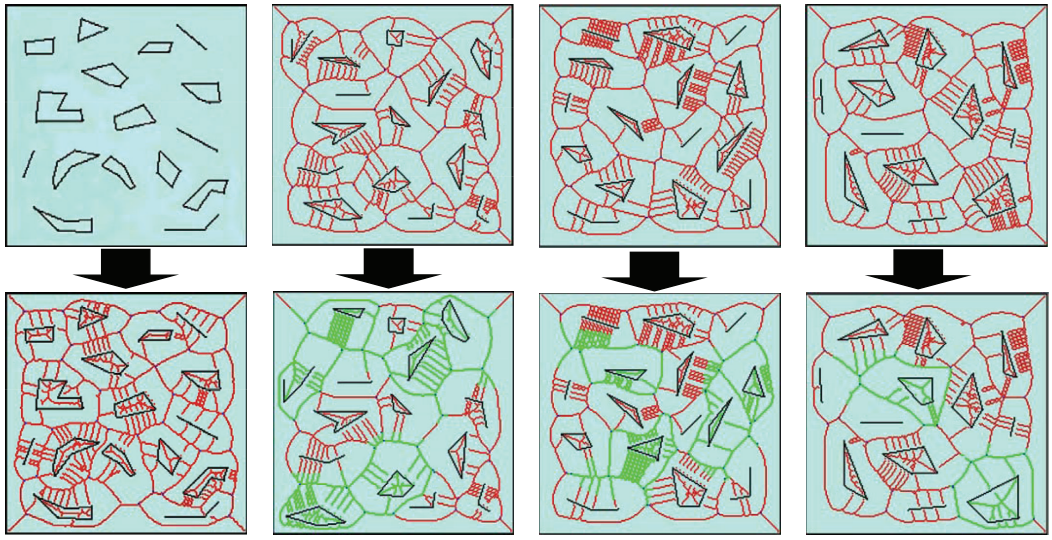
From the experiments, we further conclude that (1) for the same environment settings, the computation time of our algorithm increases in proportion to the amount of local changes, which is also true for most incremental repair algorithms; (2) for the same amount of local changes, the larger the affected area is, the more computation time will be required. In other words, changes within sparse areas spend more computation time than changes within dense areas.

5.2. Application Test for High-Level Path Planning. In order to demonstrate the usefulness of our algorithm on high-level spatial reasoning tasks, three mobile robots operating in a



(a) (b) (c)

Figure 7: The graphic representation of updating GVD vertices. Blue grids denote vertices and red ones denote edge cells.



(a) static (b) 75% moving sites (c) 50% moving sites (d) 25% moving sites

Figure 8: The maps used to test different algorithms. From left to right: the complete static map (a) which has no precomputed GVD information but only static sites within it, the maps with 75% (b), 50% (c), and 25% (d) moving sites. For each scenario, the upper map denotes the initial setting; the nether map denotes constructed or repaired GVD caused by the position or shape changes. For all maps, red girds represent original GVD edges; green grids represent the rebuilt area.

grid map of size 1001 × 501 (as shown in Figure 11) were simulated. These robots were all located at the same start grid on the bottom left. For each searching task, each robot was given a unique destination within the top right area. The searching spaces adopted by these agents were (1) the whole grid map, (2) GVD matrices generated by BL, and (3) the topological information which is generated by DTD and stored in the data structure mentioned above.

The simulation results are shown in Table 4. Agent adopting A^* to search in the whole map spends the most computation time and cell visits. Moreover, because there is no further information about maximal clearance to the sites, the resulting path (in blue) contains several cells near the sites, which will lead collisions when the physical size of the agent exceeds the limited clearance.

The agent that adopts A^* to search in the GVD matrices only explores GVD edge cells, so it saves significantly more computation time. The resulting path (in yellow) consists of (1) an initial route from the start cell to the nearest GVD cell, (2) a set of connecting GVD edges ensuring the reachability of the departure GVD cell which is nearest to the destination, and (3) a final route from departure cell to the destination.

Although the GVD matrices endowed the robot with a reduced search space, the whole process was still carried out quantitatively. In human nature, we commonly first demonstrate the connectivity of certain landmarks between the start cell and the destination. Then the detailed path planning tasks can be localized and carried out in order. Therefore, unlike the agent searching in GVD matrices, the agent searching in the GVD topologies first located the two nearest GVD vertices from start and end grids via looking up *vMap*. Then it can find out a series of connecting vertices (i.e., the landmarks) by iteratively searching connected instances in *vMap* and *eMap*. Based on these landmarks, the metric path planning task can be localized into subtasks between interconnected vertices (i.e., GVD edges storing in *eMap*). From the data shown in Table 4, we see that searching on the GVD topologies needs even less time and fewer cell visits. Moreover, an agent using DTD does not have to plan a whole metric path before moving. It plans current segment

Table 2: A comparison of the performances of the four approaches on four GVD construction and local repair scenarios: static construction for a complete map (static) and incremental construction with 75%, 50%, and 25% of the sites moving randomly in the map.

Algorithm	Static		75% move		50% move		25% move	
	μ	σ	μ	σ	μ	σ	μ	σ
Brushfire	0.4532	0.033	0.4782	0.011	0.4892	0.025	0.4633	0.012
Dynamic Brushfire*	0.4621	0.014	0.2453	0.012	0.1375	0.006	0.0249	0.011
BL*	0.4574	0.002	0.2291	0.005	0.1168	0.031	0.0117	0.068
DTD*	0.4594	0.006	0.2402	0.021	0.1172	0.104	0.0123	0.017

*Dynamic method that updates only the affected parts of the map.

Table 3: A comparison of the cell visits on four GVD construction and local repair scenarios: static construction for a complete map (static) and incremental construction with 75%, 50%, and 25% of the sites moving randomly in the map.

Algorithm	Static		75% move		50% move		25% move	
	μ	σ	μ	σ	μ	σ	μ	σ
Dynamic Brushfire	49546	1203	39732	736	22098	881	16350	903
BL	19819	864	15260	917	10278	697	5173	624
DTD	20919	791	14936	991	10662	1012	5248	471

Only local repair methods are compared.

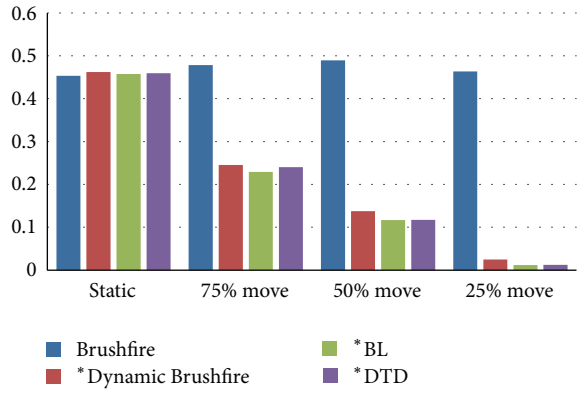


Figure 9: The average computation time for updating GVD diagrams compared to related work. The methods preceded by "∗" are dynamic methods that updates only the affected parts of the map.

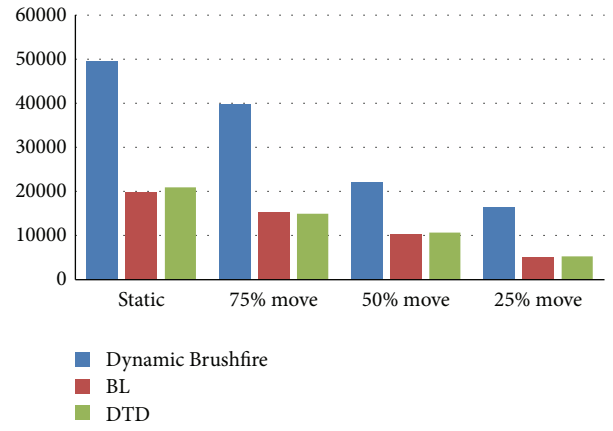


Figure 10: The average cell visits for updating GVD diagrams compared to related work.

and then check the connectivity of the next one via looking up the GVD topologies. If the underlying environment is changed, a reconstruction will be carried out by DTD and newly repaired topologies will ensure the agent replanning a high-level route to follow. Comparing to the pure metric rebuilding, such a coarse-to-fine searching strategy can avoid lots of unnecessary computation.

6. Conclusions

In this paper, we presented an algorithm, Dynamic Topology Detector (DTD), for detecting GVD topologies on girds. Compared to previous approaches, DTD explicitly provides connectivity among GVD edges, vertices, and sites, synchronously extracting it during the GVD construction. We compared our algorithm to other leading approaches and found that in a dynamic environment, our algorithm is more efficient, needing fewer cell visits than static approach. As for the competing methods which are capable of local repair, DTD can further detect the topological maps. We further demonstrated the usefulness of the resulting topologies in high-level path planning, which is particularly applicable to robots that work in areas.

Table 4: A comparison of average cell visits and execution time for the instance shown in Figure 11.

Search space	Time (second)	Cell visits	Trajectory color
Whole Map	106.377	4003426	blue
GVD matrices	0.01443	4470	yellow
GVD topologies	0.00157	1143	green

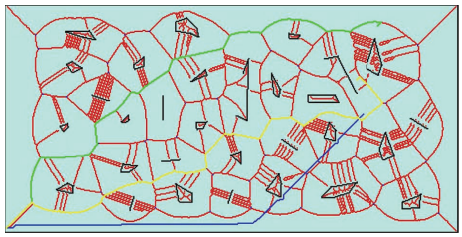


Figure 11: The graphical representation of three sample resulting paths generated by three robots. These paths are in blue (generated by searching the whole map), yellow (generated by searching GVD matrices), and green (generated by searching topological information); sites are in black, and the GVD map on this environment is in red (edges) and blue (vertices).

Acknowledgments

The authors appreciate the proofreading given by Dr. J. G. Stell. The authors appreciate fruitful discussion with the Sim812 group: Xiaocheng Liu, Shiguang Yue, Lin Sun, Qi Zhang, and Liang Zhu. Finally, The authors appreciate feedback from our reviewers.

References

[1] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: any-angle path planning on grids," *Journal of Artificial Intelligence Research*, vol. 39, pp. 533–579, 2010.

[2] Y. Lu, X. Huo, and P. Tsiotras, "Beamlet-like data processing for accelerated path-planning using multiscale information of the environment," in *Proceedings of the 49th IEEE Conference on Decision and Control (CDC '10)*, pp. 3808–3813, December 2010.

[3] M. T. Rantanen and M. Juhola, "Using probabilistic roadmaps in changing environments," *Computer Animation and Virtual Worlds*, 2013.

[4] N. M. Wardhana, H. Johan, and H. S. Seah, "Enhanced waypoint graph for surface and volumetric path planning in virtual worlds," *The Visual Computer*, vol. 29, no. 10, pp. 1051–1062, 2013.

[5] J. O. Wallgrün, "Qualitative spatial reasoning for topological map learning," *Spatial Cognition and Computation*, vol. 10, no. 4, pp. 207–246, 2010.

[6] B. Lau, C. Sprunk, and W. Burgard, "Improved updating of euclidean distance maps and Voronoi diagrams," in *Proceedings of the 23rd IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '10)*, pp. 281–286, Taipei, Taiwan, October 2010.

[7] N. Kalra, D. Ferguson, and A. Stentz, "Incremental reconstruction of generalized Voronoi diagrams on grids," *Robotics and Autonomous Systems*, vol. 57, no. 2, pp. 123–128, 2009.

[8] J. Barraquand and J.-C. Latombe, "Robot motion planning: a distributed representation approach," Tech. Rep. STAN-CS-89-1257, Computer Science Department, Stanford University, Stanford, Calif, USA, 1989.

[9] A. G. Cohn and J. Renz, "Qualitative spatial representation and reasoning," in *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz, and B. Porter, Eds., vol. 3, pp. 551–596, Elsevier, 2008.

[10] O. Takahashi and R. J. Schilling, "Motion planning in a plane using generalized Voronoi diagrams," *IEEE Transactions on Robotics and Automation*, vol. 5, no. 2, pp. 143–150, 1989.

[11] H. Choset and K. Nagatani, "Topological simultaneous localization and mapping (SLAM): toward exact localization without explicit localization," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 2, pp. 125–137, 2001.

[12] A. Franchi, L. Freda, G. Oriolo, and M. Vendittelli, "The sensor-based random graph method for cooperative robot exploration," *IEEE/ASME Transactions on Mechatronics*, vol. 14, no. 2, pp. 163–175, 2009.

[13] S. Garrido, L. Moreno, D. Blanco, and P. Jurewicz, "Path planning for mobile robot navigation using Voronoi diagram and fast marching," *International Journal of Robotics and Automation*, vol. 2, no. 1, pp. 42–64, 2011.

[14] L. Wu, M. A. Garcia, D. Puig, and A. Sole, "Voronoi-based space partitioning for coordinated multi-robot exploration," *Journal of Physical Agents*, vol. 1, no. 1, pp. 37–44, 2007.

[15] R. Fabbri, L. da F. Costa, J. C. Torelli, and O. M. Bruno, "2D Euclidean distance transform algorithms: a comparative survey," *ACM Computing Surveys*, vol. 40, no. 1, article 2, 2008.

[16] N. Rao, N. Stoltzfus, and S. S. Iyengar, "A "retraction" method for learned navigation in unknown terrains for a circular robot," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 5, pp. 699–707, 1991.

[17] H. Choset, S. Walker, K. Eiamsa-Ard, and J. Burdick, "Sensor-based exploration: Incremental construction of the hierarchical generalized Voronoi graph," *International Journal of Robotics Research*, vol. 19, no. 2, pp. 126–148, 2000.

[18] C. M. Gold, P. R. Remmele, and T. Roos, "Voronoi methods in GIS," in *Algorithmic Foundations of Geographic Information Systems*, vol. 1340, Springer, Berlin, Germany, 1997.

[19] I. Lee and M. Gahegan, "Interactive analysis using Voronoi diagrams: algorithms to support dynamic update from a generic triangle-based data structure," *Transactions in GIS*, vol. 6, no. 2, pp. 89–114, 2002.

[20] K. E. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha, "Fast computation of generalized Voronoi diagrams using graphics hardware," in *Proceedings of the 16th Annual Symposium on Computational Geometry (SIGGRAPH '99)*, pp. 375–376, 1999.

[21] L. Vachhani, A. D. Mahindrakar, and K. Sridharan, "Mobile robot navigation through a hardware-efficient implementation for control-law-based construction of generalized Voronoi diagram," *IEEE/ASME Transactions on Mechatronics*, vol. 16, no. 6, pp. 1083–1095, 2011.

[22] S. Scherer, D. Ferguson, and S. Singh, "Efficient C-space and cost function updates in 3D for unmanned aerial vehicles," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '09)*, pp. 2049–2054, Kobe, Japan, May 2009.

[23] T. Zhang and C. Suen, "A fast parallel algorithm for thinning digital patterns," *Communications of the ACM*, vol. 27, no. 3, pp. 236–239, 1984.

[24] D. Portugal and R. P. Rocha, "Retrieving topological information for mobile robots provided with grid maps," *Agents and Artificial Intelligence Communications in Computer and Information Science*, vol. 358, pp. 204–217, 2013.

[25] M. Gahegan and I. Lee, "Data structures and algorithms to support interactive spatial analysis using dynamic Voronoi diagrams," *Computers, Environment and Urban Systems*, vol. 24, no. 6, pp. 509–537, 2000.

[26] B. Lau, C. Sprunk, and W. Burgard, "Efficient grid-based spatial representations for robot navigation in dynamic environments," *Robotics and Autonomous Systems*, vol. 61, no. 10, pp. 1116–1130, 2012.