Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2014, Article ID 456739, 14 pages
http://dx.doi.org/10.1155/2014/456739

Research Article

Incremental Construction of Generalized Voronoi Diagrams on Pointerless Quadtrees

Quanjun Yin, Long Qin, Xiaocheng Liu, and Yabing Zha

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

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

Received 21 October 2013; Revised 5 January 2014; Accepted 6 January 2014; Published 26 February 2014

Academic Editor: Piermarco Cannarsa

In robotics, Generalized Voronoi Diagrams (GVDs) are widely used by mobile robots to represent the spatial topologies of their surrounding area. In this paper we consider the problem of constructing GVDs on discrete environments. Several algorithms that solve this problem exist in the literature, notably the Brushfire algorithm and its improved versions which possess local repair mechanism. However, when the area to be processed is very large or is of high resolution, the size of the metric matrices used by these algorithms to compute GVDs can be prohibitive. To address this issue, we propose an improvement on the current algorithms, using pointerless quadtrees in place of metric matrices to compute and maintain GVDs. Beyond the construction and reconstruction of a GVD, our algorithm further provides a method to approximate roadmaps in multiple granularities from the quadtree based GVD. Simulation tests in representative scenarios demonstrate that, compared with the current algorithms, our algorithm generally makes an order of magnitude improvement regarding memory cost when the area is larger than $2^{10} \times 2^{10}$. We also demonstrate the usefulness of the approximated roadmaps for coarse-to-fine pathfinding tasks.

1. Introduction

In robotics, constructing a sparse, adequate, and well-organized space model of the working area is a key issue in the successful design of a mobile robot. With such an internal description of the environment, most spatial reasoning tasks (such as path planning, self-localization, and collision detection) become feasible.

Commonly used representations for representing the environment include (but are not limited to) uniform [1] and nonuniform grid maps [2], probabilistic roadmaps [3], way point graph [4], and Generalized Voronoi Diagrams (GVDs) that are built on continuous or discrete environments. In this paper we focus on GVDs constructed on grids because of the prevalence of grid-based environment representations in mobile robot navigation [5]. GVD is defined as the set of points in free space to which the two closest sites have the same distance. 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

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

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

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

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 [6]. Due to the prevalence of grid-based environment representations in robotics, GVDs built on discrete environments are widely used and outperform other representations in extracting sparse but adequate environment skeletons [5, 7].

The advantages of employing grid-based GVDs 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 on discrete environments has drawn significant attention in recent years; among them the Brushfire algorithm and its improved versions (i.e., Dynamic

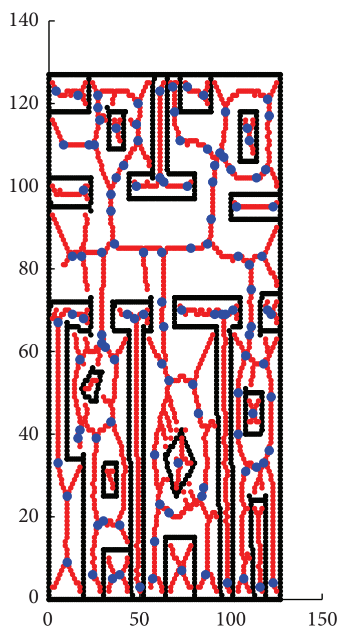


Figure 1: The GVD of an indoor environment: red lines denote the GVD edges, blue dots are GVD vertices, and black polygons represent walls and furniture, that is, the GVD sites.

Brushfire [8] and a novel approach proposed by Lau et al. in [9]) are the most representative ones.

Although these algorithms provide reduced search space, they still suffer from memory complexity. These algorithms must set aside several metric matrices to compute and maintain GVD data; therefore the memory cost depends on the resolution of the maps rather than the complexity of the space configurations. For instance, when the size of the working area enlarges from $2^N \times 2^N$ to $2^{N+1} \times 2^{N+1}$ (where $N > 0$), that is, the granularity gets finer or the side length doubles, the memory cost will be quadrupled. In addition, because these matrices are represented in single granularity, it is difficult to provide mobile robots a hierarchical data structure for carrying out coarse-to-fine navigation tasks. Although searching the finest solution is sufficient, approximating a coarser but good enough path in one higher level can be a better choice since it is time saving and is more flexible for real-time applications.

Because the GVD regions normally have strong spatial coherences, we conjecture that using quadtrees to compute and maintain GVDs can help overcome the difficulties encountered before. Based on the principle of recursive decomposition of space, cells in the metric matrices with the same value can be maximally represented as one leaf node in the quadtree structure [10]; therefore significant memory saving and a multilayered data structure can both be archived.

There are mainly two types of quadtree representations: pointer-based [11] and pointerless [12, 13]. The pointer-based quadtree is the most natural way to represent a quadtree structure. However, for very complex spatial data, the extra cost for storing pointers will exceed the amount of available memory. Some researchers tried to address this issue via employing a heuristic algorithm to find a "proper" root, so the number of the leaf nodes can be reduced [14]. However, choosing a "proper" root is unsuitable for dynamic environment, since partial changes will frequently relocate the root and thus lead a reconstruction of the whole tree. Consequently, considerable attention is concentrated on another quadtree representation method, that is, pointerless quadtree representation. This kind of representation saves more space because it does not maintain parent-to-child pointers. Via defining each tree node as a unique index, they can be maintained in a hash table through which efficient random access can be ensured.

In this paper, we design an algorithm named pointerless quadtree based GVD builder (PQ-GVD builder) which intends to achieve the following goals.

(1) Constructing GVDs on discrete environments, for example, occupancy maps.

(2) Possessing a local repair mechanism which makes use of precomputed result to update local changes efficiently.

(3) Maintaining a memory saving data structure to handle large areas or areas of high resolution efficiently.

(4) Providing approximated roadmaps in multiple granularities with which a mobile robot can carry out pathfinding tasks in a coarse-to-fine manner.

PQ-GVD builder employs a spatial hashing technique to encode quadtree nodes, defining for each of them a unique index. Therefore a quadtree can be mapped into a hash table which provides random access to any arbitrary node. Based on this encoding principle, we design PQ-GVD builder to use pointerless quadtrees in place of metric matrices to compute and maintain GVD data. Pointerless quadtrees can be memory saving, since notable entries in a matrix are merged into one leaf node if they have the same value. Moreover, rather than the depth first search manner, PQ-GVD builder traces backwards when an instant query failed (i.e., the target node is included in a high-level leaf node). This backtracking strategy is more efficient since it starts from a location which is much closer to the target node. PQ-GVD builder further provides a prudential method to approximate roadmaps in different granularities. These roadmaps are very useful for mobile robots to execute layered navigation tasks. We compare our algorithm to existing algorithms on several simulated scenarios. Our results show that when the resolution of the underlying maps is larger than $2^{10} \times 2^{10}$, the resulting GVDs outperform its grid-based counterparts in terms of less memory cost. As for the case of smaller than $2^{10} \times 2^{10}$, the number of nodes in the quadtree is not significantly less than the number of cells in the corresponding grids when representing the same GVD, so there is extra memory cost in quadtree based GVD. However, because the GVD itself has a small size and does not take up a lot of memory, the additional memory can be negligible. We further demonstrate the usefulness of the approximated roadmaps on coarse-to-fine pathfinding tasks.

The remainder of this paper is as follows: Section 2 discusses related techniques for GVD construction; Section 3

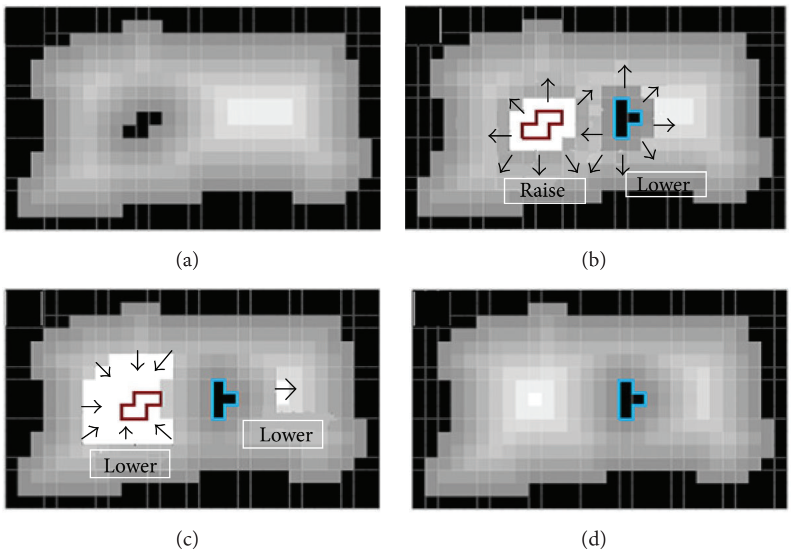


Figure 2: Distance map update between two configurations (a) and (d). Occupied cells are marked by black cells; brightness increases with distance. The inserted site (blue) initiates a "lower" wavefront shown in the intermediate steps (b) and (c) that updates the distances in the cells up to the point 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.

gives the details of the defined pointerless quadtrees; Section 4 gives the improved GVD constructing algorithm; Section 5 compares the algorithm to other algorithms and tests the usefulness of the roadmaps when used to carry out multilayer pathfinding tasks. This paper ends with conclusions in Section 6.

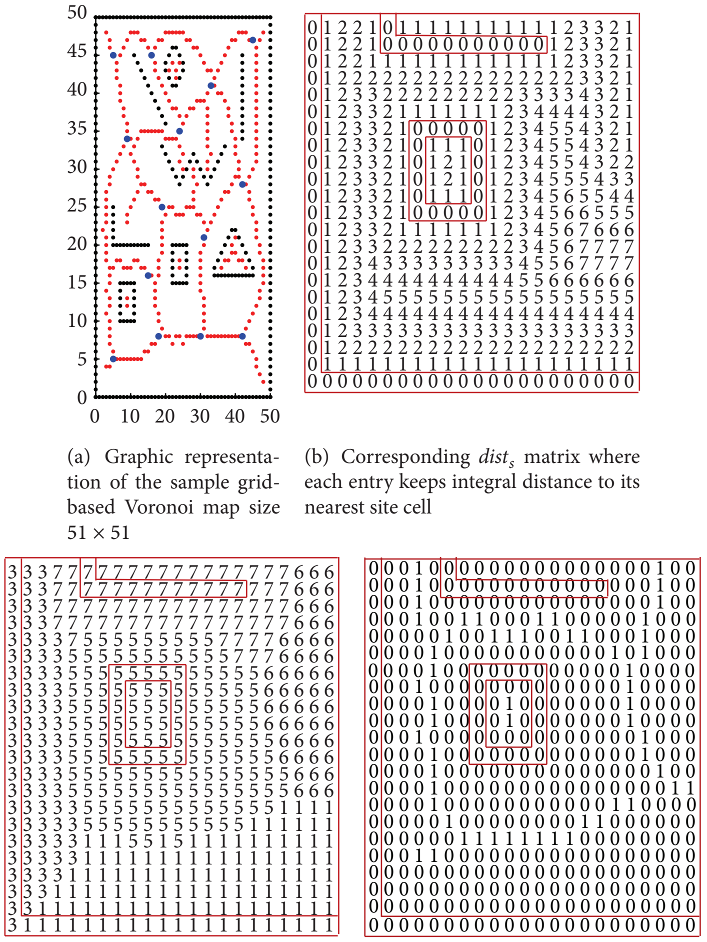
2. Related Work

Existing algorithms for computing GVDs can be roughly divided into two kinds which operate on continuous and discrete space, respectively [15]. GVDs upon continuous space are built as 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 grids. Moreover, discretizing the continuous GVD to a grid map does not work because (1) different GVD edges within the same grid cell will be mixed; (2) edges that coincidently lie between two grids can lead either two cell wide edges or invalid detection. Based on the above reasons we focus on GVDs which are computed in discrete space, that is, on grids.

As for discrete GVDs, some researchers prefer fast computation using graphics hardware [20, 21]. However, this is infeasible for (1) robots with limited hardware load in real world scenarios and (2) computer generated agents performing spatial reasoning tasks in virtual reality. Therefore many attentions concentrate on hardware-independent methods. Some of the recent approaches to rebuild GVDs on grids are based on the well-known Brushfire algorithm [22]. Brushfire is based on D* for pathfinding; it processes a priority queue *open* of the cells to propagate the change. The priority of a cell in *open* (denoted as 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 one 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 2 [7]. This wavefront updates GVD matrixes 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 [5] to incrementally rebuild GVDs on grids. In this algorithm, the entry value of $dist_s$ is estimated by gird steps accumulated throughout the propagation. Such an approximation potentially leads to 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 [23]. Following such an improvement, Lau et al. in their work proposed novel methods to rebuild GVDs with less computation time and fewer cell visits [9]; different from Kalra's work, their approach does not rely on site identifiers to detect GVD edges, so edges in the interior of a concave site can be also detected. Furthermore, Lau et al. provided additional thinning steps using "thinning patterns" proposed by Zhang and Suen [24] to get one-cell wide edges. Therefore the resulting edges are preferable in sparseness.

As shown in Figure 3, these algorithms commonly maintain three matrices, $obst_s$, $dist_s$, and $voro_s$, to represent a GVD. The matrix $dist_s$ keeps discrete or actual 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



(a) Graphic representation of the sample grid-based Voronoi map size 51 × 51

(b) Corresponding $dist_s$ matrix where each entry keeps 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 if the site cell belongs to the GVD (register as 1) or not (register as 0)

Figure 3: Grid-based GVD matrices constructed by preproposed algorithms. (a) is the resulting GVD; (b), (c), and (d) are metric matrices representing the left bottom 1/4 part.

coordinate of the exact site cell to which s is currently closest; the matrix $voro_s$ is a Boolean matrix which indicates whether s is a GVD cell.

Although these algorithms are fast and efficient, it is observed that the memory cost required by these matrices will be prohibitive when the underlying space is very large or is represented in very high resolutions. In addition, navigation tasks cannot maximally benefit from these single-layered matrixes if there are coarser but good enough solutions. Fortunately, this drawback can be remedied if a hierarchical GVD representation can be efficiently built, since one can carry out navigation tasks in a coarse-to-fine manner rather than always searching in the finest granularity. Imma Boada et al. proposed a novel approach for building polygonal approximations of GVDs that are computed in continuous space [25]. They first use quadtree to encode all the algebraic information required for generating an explicit representation of the GVD boundaries. Then, by using this hierarchical data structure a reconstruction strategy creates the multilayered approximate GVD. However, because the approximation process is mainly based on using line segments in place of curves in leaf nodes, this algorithm cannot be directly applied in discrete environments in which the sites are represented as sets of cells, not curves.

3. Pointerless Quadtree Representations

Whether a quadtree is adequate to be adopted by the proposed GVD builder depends on its efficiency on executing basic operations which will be frequently used during the construction process (i.e., updating or querying the value of a tree node efficiently). Therefore in this section we focus on designing a data structure that facilitates these operations. We first employ a spatial hashing technique proposed by Lu et al. [26] to encode each tree node as a triple. These triples will be used as indexes to map their corresponding tree nodes in a

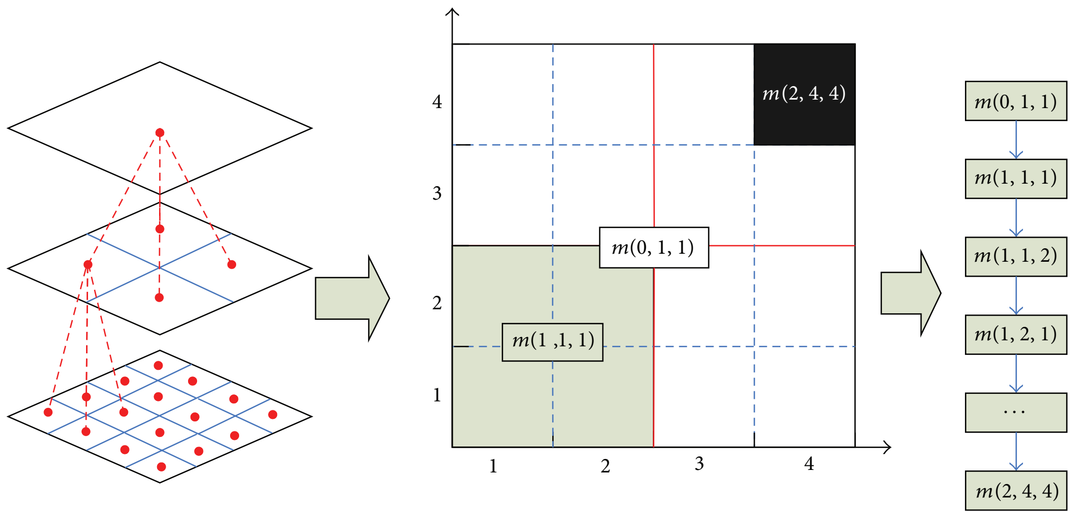


FIGURE 4: Graphic representation of the mapping process. Nodes in different layer are encoded as a unique index. Based on this encoding principle, all tree nodes can be maintained in a hash table.

hash table. We then describe the operations for updating and querying tree nodes which will be used in Section 4 by our PQ-GVD builder to compute quadtree based GVDs.

3.1. Encoding Principles. The construction of a quadtree can be carried out in a top-down fashion: a squared map is partitioned into 2×2 smaller squares; then the partition is repeated to the nonleaf squares until the child squares are pure or the finest resolution is reached. The directional path for a tree node m in layer s is represented as $m(s; x_m, y_m)$, where $0 \le s \le N$ denotes the current layer and (x_m, y_m) denotes the coordinates of m. m is a dyadic partition that contains a set of the finest cells in layer N:

$$m(s; x_m, y_m) = \left\{(N; i, j) : 2^{N-s}(x_m - 1) + 1 \le i \le 2^{N-s} x_m, \; 2^{N-s}(y_m - 1) + 1 \le j \le 2^{N-s} y_m\right\}. \tag{3}$$

In virtue of the hierarchical structure nature, when a tree node $m(s; x_m, y_m)$ is given, we can obtain its father node as $f(s-1; [x_m/2]+1, [y_m/2]+1)$, where the symbol "$[*]$" means taking the lower integer value. We can also partition m into four child nodes in layer $s+1$; that is,

$$\begin{aligned} m(s; x_m, y_m) = \{ & c_1(s+1; 2x_m - 1, 2y_m - 1) \\ & \cup c_2(s+1; 2x_m, 2y_m - 1) \\ & \cup c_3(s+1; 2x_m, 2y_m) \\ & \cup c_4(s+1; 2x_m - 1, 2y_m)\}. \end{aligned} \tag{4}$$

For convenience, in the remainder of this paper we represent the partition operation on node m as $m.child$ and its father node will be denoted as $m.father$.

Figure 4 shows a simple example of mapping a quadtree into a hash table. The value of a node can be one of the two distinct nonnegative integers:

$$m.state = \begin{cases} 0 & \text{for inner node} \\ c & \text{for leaf node,} \end{cases} \tag{5}$$

where $c > 0$ is a positive integer, representing the type of the node. Compared to the standard pointer-based quadtree description that requires six entries for the inner nodes (i.e., five pointers referring to the father and child nodes and one entry maintaining the value), in this case only four integers are needed (i.e., one for layer, two for coordinates, and one for value). Hence, more than 33% memory can be saved.

3.2. Algorithms for Updating and Querying Tree Nodes. Different from the depth first searching manner adopted by pointer-based quadtrees, our algorithm accesses the target node by referring to its index. If the node has not yet been built, its nearest ancestor will be located via launching a backtracking process. Algorithm 1 shows the pseudocode of the algorithm to update and query a tree node. Initially, the quadtree is free and thus there is only one leaf node, that is, the root, in the hash table. During the process of updating, new nodes will be created and inserted. If the four child nodes with regard to the same parent node possess the same state value owing to an updating operation, a backtracking will be launched to merge these child nodes, modifying their father's state value into a leaf node and erasing these child nodes from the hash table.

In the function of **UpdateANode**, if a tree node $m(s; x_m, y_m)$ has already been inserted into the table (lines 1 to 13), then its state will be updated through an instant access. After the update, if m is a leaf node, a backtracking process will iteratively merge child nodes that have the same value (lines 5 to 9). This process will be terminated when an inner node (i.e., condition in line 5 is not satisfied) or the root of

```
UpdateANode ($m(s; x_m, y_m)$, state)
1.  if m exists in the hash table
2.    if $m.state \neq 0$ (i.e., m is a leaf node)
3.      $m.state \leftarrow state$
4.      $n \leftarrow m.father$
5.      while all n's children possess the same state
6.        erase all n's children from the table
7.        $n.state \leftarrow state$, $n \leftarrow n.father$
8.        if n is the root node
9.          break the while loop
10.   if $m.state = 0$ (i.e., m is a inner node)
11.     $m.state \leftarrow state$
12.     for each $n \leftarrow m.child$
13.       ClearTree (n)
14. if m doesn't exist in the hash table
15.   NewANode ($m(s; x_m, y_m)$, state)
ClearTree ($n(s; x_n, y_n)$)
16. if $n.state = 0$ (i.e., n is a inner node)
17.   $m \leftarrow$ for each n.child
18.     ClearTree (m)
19. else erase n from the table
NewANode ($m(s; x_m, y_m)$, state)
20. while m doesn't exist and $m \neq$ root node
21.   $m \leftarrow m.father$
22. if $m.state \neq state$
23.   $m.child \leftarrow$ new 4 leaf nodes for
24.   for each $n \leftarrow m.child$
25.     if $(x_n, y_n) = (x_m, y_m)$ $n.state \leftarrow state$
26.     else $n.state \leftarrow m.state$
27.   $m.state \leftarrow 0$
QueryANode ($m(s; x_m, y_m)$)
28. while m doesn't exist and $m \neq$ root node
29.   $m \leftarrow m.father$
30. return m.state
```

Algorithm 1: Pseudocode for updating and querying a node state value in the pointerless quadtree.

the tree is reached (line 9). Otherwise if m is an inner node, its offspring will be erased from the table after its state is updated (lines 10 to 13). These erasing operates are finished by function **ClearTree** (lines 16 to 19).

In the contrast, if there is no record of node m in the table, then the function **NewANode** will be called to create corresponding node instances (lines 14 to 15). It is clear that no record of m in the table means that an ancestral leaf node that contains the square area of m should be first located in function **NewANode** (lines 20 to 21). After the location, if this ancestor has a different value, its four child nodes will be created and the state of the ancestor will become 0 (converts to an inner node) (lines 22 to 27). The searching mechanism of the function **QueryANode** is analogous to **NewANode** (lines 28 to 30).

4. Incrementally Constructing GVDs in Pointerless Quadtrees

Existing approaches for incrementally constructing GVDs commonly set aside metric matrixes (as shown in Figure 3) to maintain GVD data and update them during the propagation of wavefronts. The main disadvantage of using these matrixes is that the required memory is prohibitive when the granularity is getting finer or the environment is very large. Based on the pointerless quadtree representation discussed in Section 3, we in this section use the pointerless quadtree in place of preused metric matrixes to maintain corresponding data.

Table 1 shows the GVD data which will be processed during the execution of our algorithm. Those with symbol "*" are maintained in pointerless quadtrees. We do not use quadtree to represent $Obst_s$ because its contour feature does not provide strong spatial coherences and representing it in quadtree will be more expensive than in metric matrix.

Figure 5 shows the flowchart describing the main steps of the algorithm. The update is triggered by events which makes some cells in the grid map 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** and (or) **FreeSiteCell**, all changed grid cells are inserted into the priority queue *Open* which is sorted by the entries in *Dist*. In step 2, function **UpdateGVD**

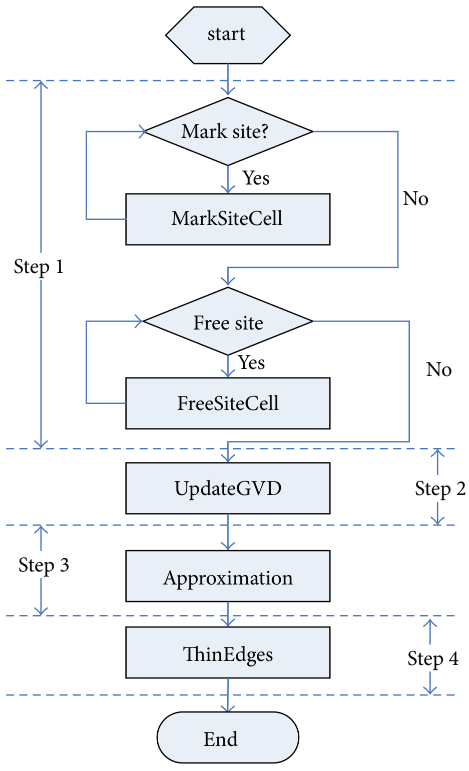


Figure 5: The flowchart describing the process of building a quadtree based GVD.

Table 1: The GVD data used by PQ-GVD builder.

Data name	Semantics
**Voro*	A pointerless quadtree storing the resulting GVD indicating if a grid is a GVD edge cell (by checking its state as "true/1" or "false/2").
**Raise*	A pointerless quadtree indicating the processing type for a grid (by checking its state as "Raise/1" or "Lower/2").
**Obst*	A pointerless quadtree indicating which site a grid belongs to (by checking its state as site ID, represented in positive integer).
Dist	A 2D integer matrix storing for each grid the distance corresponding to its nearest site cell.
Open	A priority queue storing grid cells that are enqueued when the wavefront (i.e., raise or lower) propagates to them. Entries in *Open* are sorted in ascending order; cells with lower value in matrix *Dist* possess higher priority.

propagates the changes until there is no more affected cell remaining in *Open* list and thus builds the multilayered GVD. Step 3 calls the function **Approximate** to obtain pieces of coarser but good enough roadmaps from the multilayered GVD. In step 4, function **ThinningEdges** thins the rough result to get one-cell wide edges.

4.1. Updating a GVD. The initial values of the GVD data are set as $Obst = 1$, $Dsit = \infty$, $Voro =$ false, and $Raise =$ false. This is based on the fact that there is no site within the working space, or existing sites are infinitely far away. As shown in Algorithm 2, when grid cell m is marked as a site cell by calling function **MarkSiteCell**, its $Dist[x_m][y_m]$ equals 0 and refers to itself as the closest site (lines 31 to 34). Reversely, when m is freed by calling function **FreeSiteCell**, its corresponding states are reset (lines 35 to 39). Function **push**(m, d) inserts m into *Open* with priority d or updates the priority if m has already been inserted (lines 34 and 39).

In the second step, the function **UpdateGVD** orderly pops the next unprocessed cell m with the lowest *Dist* value until the queue is empty (lines 40 to 41). If m is cleared and not yet propagated a raise wavefront, the function **PorcessRaise** is called (lines 42 to 44). If m has a valid closest site, then the function **ProcessLower** is called (lines 45 to 49). Therefore a lower wavefront is propagated. The pseudocode for the raise and lower propagation is shown in Algorithm 3.

All cells enqueued in *Open* will be processed by either **ProcessLower**(s) or **ProcessRaise**(s). At the beginning, newly occupied cells call function **ProcessLower**(s) to launch a "lower" wavefront which propagates the changes of *Dist* and *Obst* to the affected cells, for example, 8-connected grids (lines 59 to 67). 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 50 to 58). During the interwoven of these two wavefronts, neighbors affected by the processed cell are again enqueued in *Open* and therefore the propagation continues.

The rough GVD edge cells are marked by calling function **CheckVoro** when the condition $d < Dist[x_n][y_n]$ in line 64 is not satisfied. **CheckVoro** first tests if at least one of the two cells (i.e., m and n) is not adjacent to its closest site. If m has a valid closest site that is different from the closest site of c, both m and n can be the edge cell candidates (lines 72 to 76).

4.2. Approximation. The concept of an approximation is an inexact representation of something that is still fine enough to be used [14]. In many real world applications, even if an accurate spatial representation is available, an approximated model can be preferable since it is good enough and can significantly minimize the amount of computation and complexity. For instance, a mobile robot can speed up its navigation if it uses an approximated space model because much fewer cell visits and computation are ensured.

In this section, we discuss how to approximate from a pointer-quadtree based *Voro* which is obtained after step 2 (i.e., **UpdateGVD** shown in Figure 5). An outer approximation principle proposed by Ranade et al. [27] is applied, treating grey nodes as GVD cells. A more accurate definition is given by Samet [28]: "Given an image I, the outer approximation, $OB(k)$ is a binary image defined by black blocks (in this paper stands for GVD cells) and the grey nodes at level k." As an example, a GVD of dimension $2^8 \times 2^8$ which divides five points is given in Figure 6(a). By executing function **UpdateGVD**(), a quadtree based *Voro* can be constructed as shown in the top left of Figure 6(b). The top right, bottom right, and bottom left denote approximations at level 7, level 6, and level 5, respectively. The pseudocode

```
MarkSiteCell ($m(s; x_m, y_m)$, $sID$)
31. $Dist\,[x_m][y_m] \leftarrow 0$
32. Obst.UpdateANode($m$, $sID$)
33. Check.UpdateANode($m$, true)
34. Open.push($m$, 0)
End
FreeSiteCell ($m(s; x_m, y_m)$)
35. Obst.UpdateANode($m$, 1)
36. Raise. UpdateANode($m$, true)
37. $Dist\,[x_m][y_m] \leftarrow \infty$
38. Check.UpdateANode($m$, true)
39.     Open.push($m$, 0)
End
UpdateGVD ()
40.     while Open is not empty
41.     $m(s; x_m, y_m) \leftarrow pop(Open)$
42.     $b \leftarrow$ Raise.QueryANode($m$)
43.     if $b$ = true
44.         ProcessRaise($m$)
45.     else
46.         $c \leftarrow$ Obst.QueryANode($m$)
47.         if $c \neq 1$
48.             Voro.UpdateANode($m$,false)
49.             ProcessLower($m$)
End
```

Algorithm 2: Pseudocode for updating a GVD.

for the function **Approximation** is shown in Algorithm 4. Because the approximations for each level take read-only operations, that is, query, upon the quadtree *Voro*, they can be processed in parallel.

The approximate Voronoi matrix is a Boolean matrix denoted as *ApproVoro* (line 86). This function checks the state values of all nodes in the level specified by the *layer* parameter. Pure GVD cell nodes and gray nodes in this level are registered as approximate GVD cells (lines 78 to 85). This approximation process can be constrained in the area affected by current update. In step 2, when each cell is popped from the *Open* list, a global variable *changeArea* will be checked, updating itself if the popped cell exceeds the area that *changeArea* currently maintains. In the approximation process, variable *changeArea* is used to check if an entry of the approximate matrix should be processed (line 80).

4.3. Thinning Edges of Approximated GVDs. The approximate GVD matrices obtained after step 3 contain rough edges which are two or three cells wide. In the step of thinning the rough edges, the thinning patterns (as shown in Figure 7) proposed by Lau et al. [7] are employed to refine these edges. The input taken by this thinning is the priority queue, *roughtEQueue*, which involves all edge cells that are newly created by **Approximation** (line 84). 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 7 matches its location, then it is redundant and can be removed by setting $ApproVoro[i][j] = false$ without destroying the connectivity.

5. Experiments and Analysis

In this section we employ statistical methods to compare our algorithm with other competing methods on several simulated scenarios. We also demonstrate the usefulness of the quadtree based GVDs when used to carry out pathfinding tasks.

In the worst case, that is, to represent a grid-based map in which each grid possesses a different state value with regard to its adjacent grids, the quadtree must expand all its inner nodes till the most precise level. Suppose the size of the map is $2^N \times 2^N$, $N = 1, 2, \ldots$; then the total size of the corresponding pointerless quadtree will be

$$1 + 4 + 4^2 + \cdots + 4^N = 1 + \left[\frac{4\left(1-4^N\right)}{(1-4)}\right]. \quad (6)$$

So the space complexity is $O(4^N)$.

As for the time complexity of querying a node, since we build the quadtree in terms of hash tables, an existing node can be located in constant time. However sometimes the node is absorbed by a larger leaf node and there is no instance inserted in the table. In this case we can search according

```
ProcessRaise ($m(s; x_m, y_m)$)
50. for all $n(s; x_n, y_n) \in$ Adj8($c$)
51.     $b \leftarrow$ Obst.QueryANode($n$)
52.     $c \leftarrow$ Raise.QueryANode($n$)
53.     if $Dist[x_n][y_n] \neq \infty$ and $c$ = true
54.         if $b$ has been removed
55.             ClearCell($n$)
56.             Raise.UpdateANode($n$, true)
57.             Open.push($n$, $Dist[x_n][y_n]$)
58.     Raise.UpdateANode($m$,false)
End
ProcessLower ($m(s; x_m, y_m)$)
59. for all $n(s; x_n, y_n) \in$ Adj8($m$)
60.     $b \leftarrow$ Raise.QueryANode($n$)
61.     $c \leftarrow$ Obst.QueryANode($m$)
62.     if $b$ = false
63.         $d \leftarrow Dist[x_m][y_m] + dist$
64.         if $d < Dist[x_n][y_n]$
65.             Obst.UpdateANode($n$,$c$)
66.             Open.push($n$, $d$)
67.         else CheckVoro ($m$,$n$)
End
CheckVoro($m(s; x_m, y_m)$, $n(s; x_n, y_n)$)
68. $a \leftarrow$ Obst.QueryANode($m$)
69. $b \leftarrow$ Obst.QueryANode($n$)
70. $c \leftarrow Dist[x_m][y_m]$
71. $d \leftarrow Dist[x_n][y_n]$
72. if $(c > 10 \vee d > 10) \wedge (a \neq 1) \wedge (a \neq b)$
73.     if $c \leq d$
74.         Voro.UpdateANode($m$, true)
75.     if $d \leq c$
76.         Voro.UpdateANode($n$, true)
End
```

Algorithm 3: Pseudocode for the raise and lower propagation.

to the layer number and the location that are included in the given index. Based on such a backtracking strategy the searching time only depends on the depth of the quadtree. In the worst case the time complexity is degenerated to $O(N)$, which equals the average time needed by a blind search.

The following experimental analysis is to show that the pointerless quadtree representation method is quite suitable for storing grid-based data structures which possess large areas of the same state value. The GVD data which is generated and maintained by the proposed PQ-GVD builder just has such kind of structural features.

5.1. Experimental Analysis. We compare our algorithm to Dynamic Brushfire (we abbreviate it as DB below) and the method proposed by Lau et al. (we abbreviate it as BL below) [9]. The scenarios are a set of $2^N \times 2^N$ grid maps (N is an integer and $7 \leq N \leq 13$) which have approximately 20% cells occupied by several predefined sites. We take Figure 8 as an example to describe the performance of different algorithms. The size of the underlying map shown in Figure 8 is $2^8 \times 2^8$. Figures 8(a), 8(b), and 8(c) are GVDs built by DB, BL, and our algorithm, respectively. Figure 8(d) to Figure 8(f) are approximated roadmaps extracted from Figure 8(c). We ran each algorithm in each granularity for 100 times.

The comparison of the memory cost among the three approaches is shown in Table 2 and Figure 9. From the table and the figure we see that, for the maps that are smaller than $2^{10} \times 2^{10}$, our PQ-GVD builder needs more space to maintain *Obst* and *Voro* than those used by DB and BL. This is because the extra memory required by a quadtree can offset or even exceed the memory our PQ-GVD builder can save. Nevertheless, as the memory cost is not prohibitive in the cases of small maps, the increased cost can be negligible (e.g., for the worst case, a $2^{10} \times 2^{10}$ map needs only 1.4 megabytes to maintain its corresponding quadtree representation). For maps that are larger than $2^{10} \times 2^{10}$, our PQ-GVD builder generally saves an order of magnitude memory, because the quadtree representations on *Obst* and *Voro* save a lot of memory by merging sets of cells that have the same values into one node. In addition, the matrix *bRaise* even needs only 32 bytes to maintain the data for any arbitrary granularity, since, when the update is terminated, all entries in the matrix will be set as false and there will be only a root node in its quadtree representation.

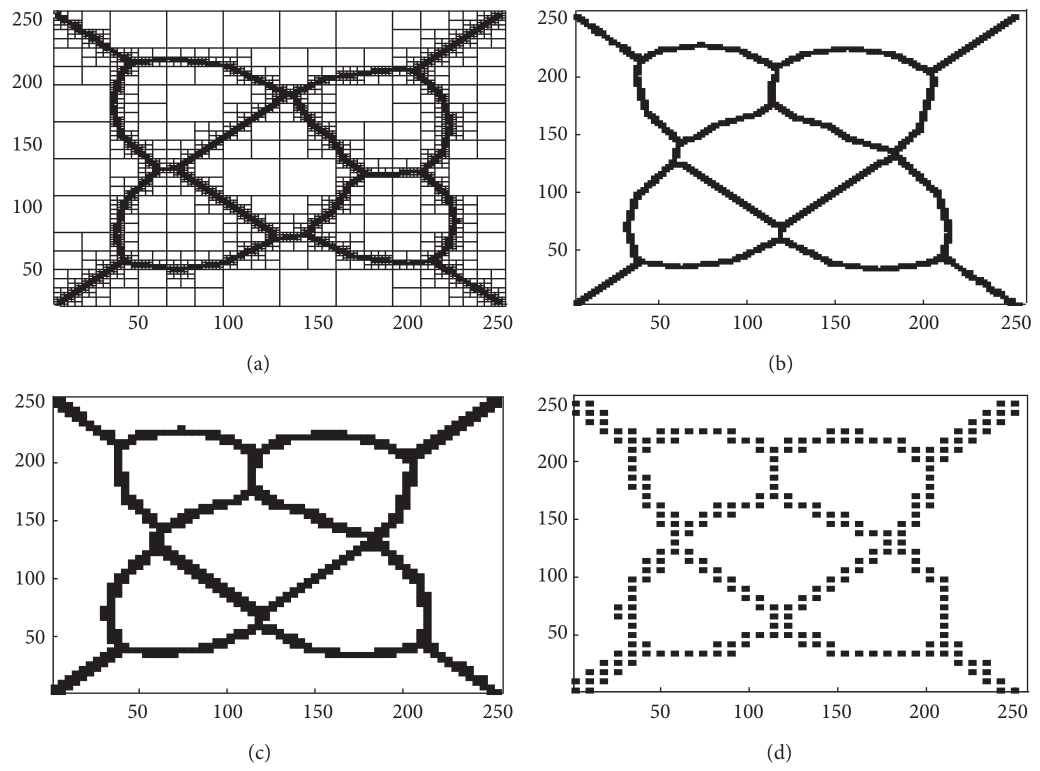


Figure 6: The original pointerless quadtree based GVD (a) and the approximations built in level 7 (b), level 6 (c), and level 5 (d), respectively.

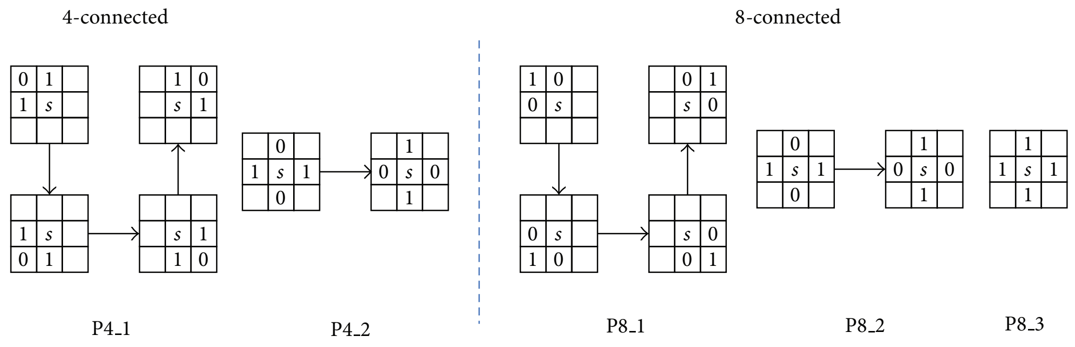
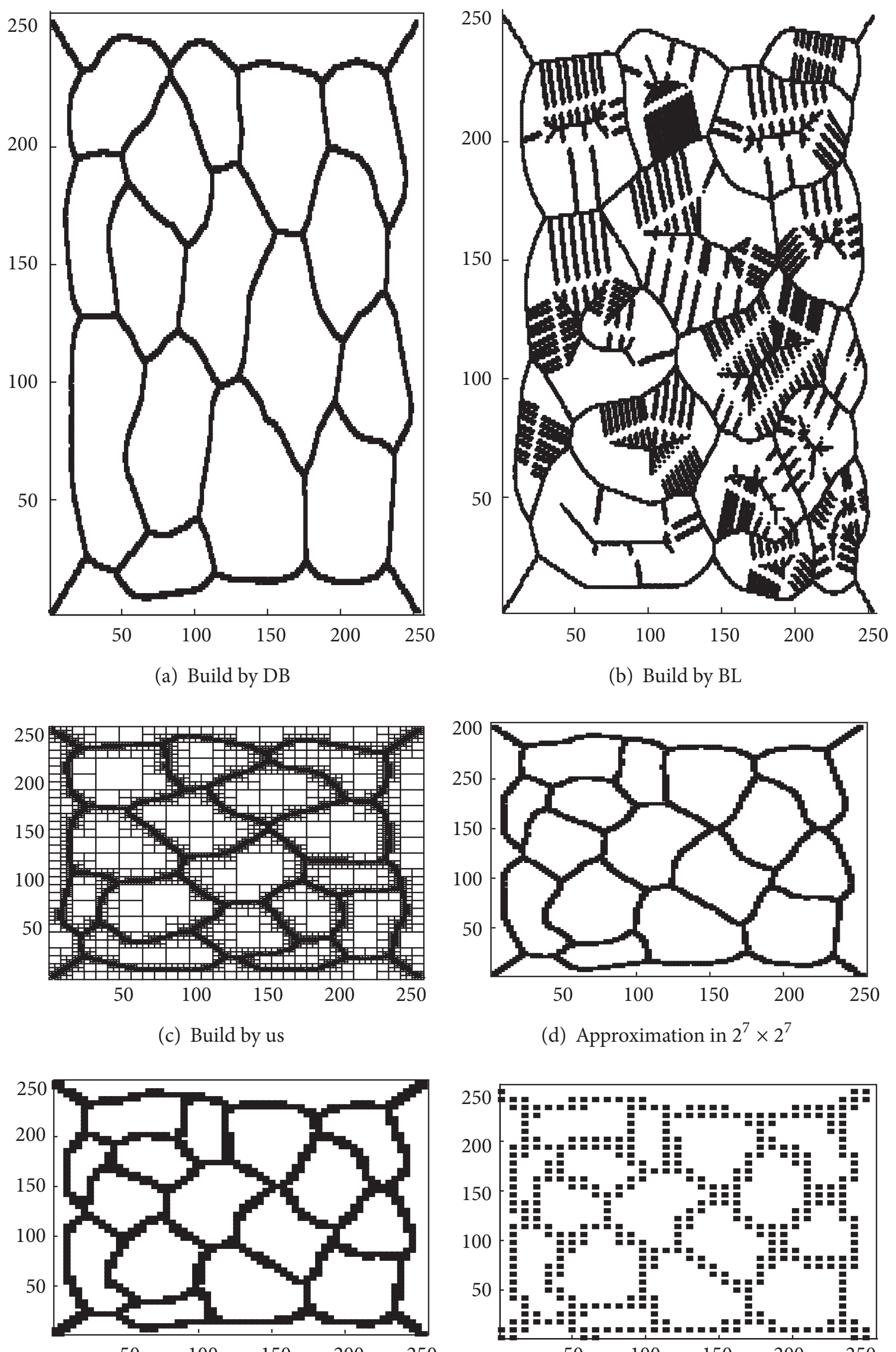


Figure 7: Patterns used by edge thinning. Arrows indicate application of rotated copies.

5.2. Usefulness in Multilayered Pathfinding Tasks. In order to demonstrate the usefulness of the quadtree based GVDs on navigation tasks, three mobile robots operating in a grid map of size $2^8 \times 2^8$ (as shown in Figure 10) 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 multilayered roadmaps which are approximated from the pointerless quadtree based GVDs.

For those pathfinding methods who do not construct GVDs, their underlying searching space is the whole grid map and the basic searching strategy is also the A* algorithm. According to the pathfinding problem requirements, some extra improvements are needed to make the algorithm be practically implementable. For example, the improved algorithms should be able to generate smooth path and carry out efficient collision detections.

There are indeed some improvement versions to deal with issues mentioned above. To improve the optimality and smooth of the resulting paths it is often necessary to apply a postprocessing smoothing step [29]; lifelong planning A* (LPA*) [30] and Dynamic A* Lite (D* lite) [31] et al. were proposed to be incremental search algorithms; as for the large search space, the Hierarchical Pathfinding A* (HPA*)



(a) Build by DB

(b) Build by BL

(c) Build by us

(d) Approximation in $2^7 \times 2^7$

(e) Approximation in $2^6 \times 2^6$

(f) Approximation in $2^5 \times 2^5$

Figure 8: The maps used to test different algorithms.

[32] is proposed to carry out a coarse-to-fine search. Besides variant versions of the basic A* algorithm, the potential fields of the map can be precomputed to generate driven forces, navigating agents heading to the destination [33].

The reason we use GVD as the underlying search space in pathfinding tasks is because: (1) when compared to the whole grid map, a GVD is a sparse skeleton and a much reduced search space; (2) the proposed PQ-GVD builder can incrementally construct and locally repair the GVD, which is quite efficient for dynamic environments; (3) a GVD generates information about maximum clearance to the obstacles in the map so collision check is easy to be carried

Table 2: A comparison of the memory cost (in megabyte) of the proposed algorithm to its counterparts. The sizes of the underlying maps are set as $2^N \times 2^N$, where N ranges from 7 to 13.

Algorithm	$N = 7$	8	9	10	11	12	13
DB							
Obst	0.06251	0.25002	1.00013	4.00001	16.00021	64.00011	256.00032
Voro	0.01562	0.0625	0.25003	1.00000	4.00005	16.00002	64.00012
bRaise	0.01561	0.0625	0.25007	1.00001	4.00000	16.00002	64.00008
BL							
Voro	0.01562	0.0625	0.25003	1.00000	4.00005	16.00002	64.00012
bRaise	0.01561	0.0625	0.25007	1.00001	4.00000	16.00002	64.00008
PQ-GVD builder							
Obst	0.080097	0.187626	0.459812	0.767155	1.900394	2.352141	6.338497
Voro	0.139286	0.316853	0.796131	1.277332	3.168629	5.103362	8.326475
bRaise	0.000034	0.000034	0.000034	0.000034	0.000034	0.000034	0.000034

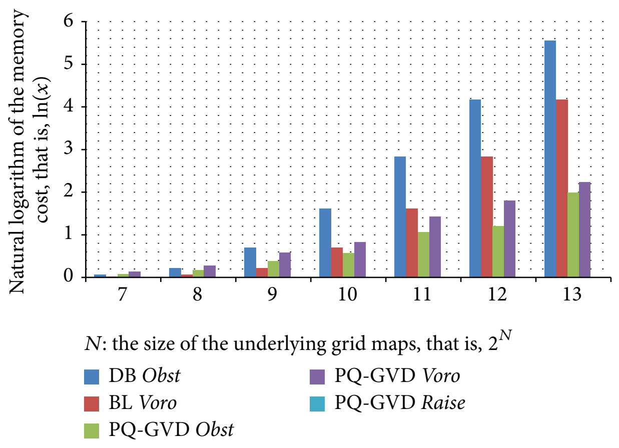


Figure 9: The average computation time for updating GVD diagrams compared to related work.

out; (4) as a by-product of the GVD builder, the distance matrix can be used to build a potential field; this field can generate artificial forces to navigate agents onto the GVD edges so that local minima near by the obstacles could be avoided efficiently.

The simulation results are shown in Figure 10 and Table 3. Agent adopting A* to search in the whole map (blue path in Figure 10(a)) 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 (green path in Figure 10(a)), so it saves significantly more computation time. The resulting path 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 the nearest to the destination; (3) a final route from departure cell to the destination.

Table 3: A comparison of average cell visits and execution time for the instance shown in Figure 10.

Search space	Time (second)	Cell visits	Trajectory color
Whole map	72.91	56326	Blue
GVD matrices	0.01443	1101	Green
Approximation in $2^7 \times 2^7$	0.00147	831	Red in Figure 10(b)
Approximation in $2^5 \times 2^5$	0.00024	373	Red in Figure 10(c)

Although the GVD matrices endowed the agent with a reduced search space, the whole process is still in one single granularity. From a practical point of view, we only need to demonstrate a coarser but good enough path in a higher level, while the detailed path can be localized and worked out in order. Therefore, unlike the agent searching in GVD matrices, the agent searching in a coarse-to-fine manner first tried to find a path from the top layer (as shown in Figure 10(b)). If it is not found, another search in the next finer layer will be executed. From the data shown in Table 3 we see that searching on the roadmaps needs even less time and fewer cell visits. Moreover, if a refined path is needed, an agent does not have to plan the whole finest path before moving. It can divide the resulting coarse path into several segments and then carry out detailed navigation just on current segment. Such a strategy is more flexible since if the underlying environment is changed, a reconstruction will be carried out and newly repaired roadmaps will ensure the agent replanning a new high-level route to follow.

6. Conclusions

In this paper we presented an algorithm named pointerless quadtree based GVD builder (PQ-GVD builder) to incrementally update GVDs in discrete environments. Compared to previous approaches, PQ-GVD builder uses pointerless quadtrees in place of metric matrices to compute and maintain GVDs. An efficient hashing technique was adopted to

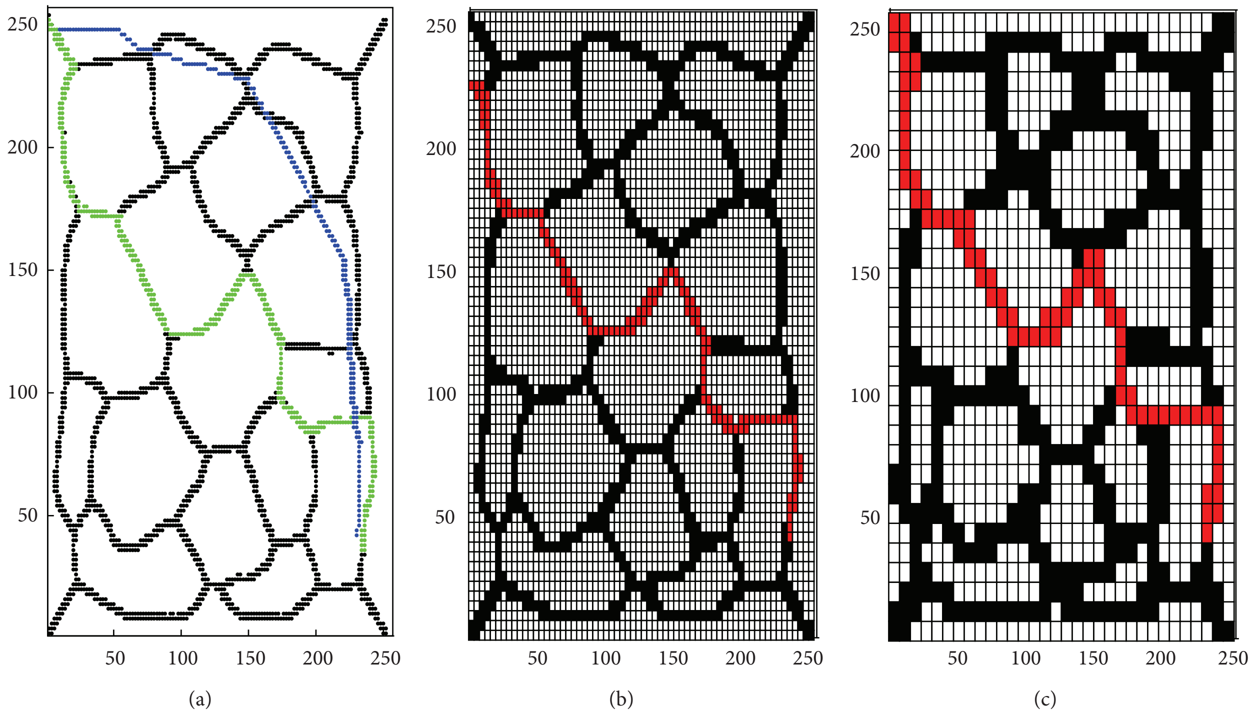


FIGURE 10: The graphical representation of three sample resulting paths generated by three robots. These paths are in blue (generated by searching the whole map in (a)), green (generated by searching GVD matrices in (a)), and red (generated by searching approximate map, that is, $2^5 \times 2^5$ in (b) and $2^7 \times 2^7$ in (c)).

```
Approximation (layer, changeArea)
77. range ← $2^{\text{layer}}$
78. for each i from 0 to range
79.     for each j from 0 to range
80.         if (i, j) is in changeArea
81. s ← Voro.QueryANode(m(layer; i, j))
82. if s.state = gray or s.sate = true
83.     b ← true
84.     roughtEQueue.push((layer; i, j))
85. else b ← false
86. ApproVoro[i][j] ← b
```

ALGORITHM 4: Pseudocode for approximations.

encode each tree node to map the whole tree into a hash table so that random access and instant query can be ensured. Based on the encoding principles, we designed a PQ-GVD builder to incrementally constructing quadtree based GVDs. The statistic test showed that the quadtree based GVDs can be used to represent large scale maps or maps with high resolutions. Due to the feature of multilayered data structure, we further proposed an approximation method to retrieve roadmaps in different granularities. These roadmaps can be adopted by mobile robots to carry out coarse-to-fine path finding tasks and are more efficient when inexact but good enough paths exist in higher levels.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China no. 91024030. The authors appreciate fruitful discussion with the Sim812 group: Peng Jiao, Xiaocheng Liu, Shiguang Yue, Lin Sun, Qi Zhang, and Liang Zhu. Finally, they appreciate feedback from their 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] 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.

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

[7] 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.

[8] 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.

[9] 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, 2013.

[10] J. Vörös, "Low-cost implementation of distance maps for path planning using matrix quadtrees and octrees," *Robotics and Computer-Integrated Manufacturing*, vol. 17, no. 6, pp. 447–459, 2001.

[11] F. Dehne, A. G. Ferreira, and A. Rau-chaplin, "Parallel processing of pointer based quadtrees on hypercube multiprocessors," in *Proceedings of the International Conference on Parallel Processing*, 1991.

[12] M. G. Choi, E. Ju, J.-W. Chang, J. Lee, and Y. J. Kim, "Linkless octree using multi-level perfect hashing," *Computer Graphics Forum*, vol. 28, no. 7, pp. 1773–1780, 2009.

[13] F. B. Atalay and D. M. Mount, "Pointerless implementation of hierarchical simplicial meshes and efficient neighbor finding in arbitrary dimensions," *International Journal of Computational Geometry & Applications*, vol. 17, no. 6, pp. 595–631, 2007.

[14] X. Yin, I. Düntsch, and G. Gediga, *Quadtree Representation and Compression of Spatial Data*, Springer, Berlin, Germany, 2011.

[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. S. V. 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," *The 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, pp. 21–35, 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 26th annual conference on Computer graphics and interactive techniques (SIGGRAPH '99)*, 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] J. Barraquand and J. -C. Latombe, "Robot motion planning: a distributed representation approach," Tech. Rep. STAN-CS-89-1257, Computer Science Department, Stanford University, 1989.

[23] 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.

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

[25] I. Boada, N. Coll, N. Madern, and J. A. Sellarès, "Approximations of 2D and 3D generalized Voronoi diagrams," *International Journal of Computer Mathematics*, vol. 85, no. 7, pp. 1003–1022, 2008.

[26] 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.

[27] S. Ranade, A. Rosenfeld, and H. Samet, "Shape approximation using quadtrees," *Pattern Recognition*, vol. 15, no. 1, pp. 31–40, 1982.

[28] H. Samet, "Data structures for quadtree approximation and compression," *Communications of the ACM*, vol. 28, no. 9, pp. 973–993, 1985.

[29] I. Millington and J. D. Funge, *Artificial Intelligence for Games*, Taylor & Francis, New York, NY, USA, 2009.

[30] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A∗," *Artificial Intelligence*, vol. 155, no. 1-2, pp. 93–146, 2004.

[31] S. Koenig and M. Likhachev, "D* Lite," in *Proceedings of the AAAI Conference of Artificial Intelligence*, pp. 476–483, August 2002.

[32] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical path-finding," *Journal of Game Development*, vol. 1, pp. 7–28, 2004.

[33] J. Hagelback, "Potential-field based navigation in StarCraft," in *Proceedings of the IEEE International Conference on Computational Intelligence and Games (CIG '12)*, 2012.