Hindawi
Mathematical Problems in Engineering
Volume 2020, Article ID 5750739, 13 pages
https://doi.org/10.1155/2020/5750739

Research Article

Hexagon-Based Generalized Voronoi Diagrams Generation for Path Planning of Intelligent Agents

Fen Tang, Xiong You, Xin Zhang, and Kunwei Li

College of Geospatial Information, Information Engineering University, Henan, Zhengzhou 450052, China

Correspondence should be addressed to Xiong You; youarexiong@163.com

Received 27 August 2019; Accepted 22 February 2020; Published 23 April 2020

Academic Editor: Ivan Giorgio

Grid-based Generalized Voronoi Diagrams (GVDs) are widely used to represent the surrounding environment of intelligent agents in the fields of robotics, computer games, and military simulations, which improve the efficiency of path planning of intelligent agents. Current studies mainly focus on square-grid-based GVD construction approaches, and little attention has been paid to constructing GVDs from hexagonal grids. In this paper, an algorithm named hexagon-based crystal growth (HCG) is presented to extract GVDs from hexagonal grids. In addition, two thinning patterns for obtaining one-cell-wide GVDs from rough hexagon-based GVDs are proposed. On the basis of the principles of a leading square-grid-based algorithm named Brushfire, a hexagon-based Brushfire algorithm is realized. A comparison of the HCG and the hexagon-based Brushfire algorithm shows that HCG is much more efficient. Further, the usefulness of hexagon-based GVDs for the path planning tasks of intelligent agents is demonstrated using several representative simulation experiments.

1. Introduction

Spatial representation is considered to be a fundamental subject in the fields of robotics, computer games, and military simulation. In applications in these fields, one or more intelligent agents frequently implement spatial reasoning tasks such as route planning, self-localisation, and collision avoidance. Generally speaking, the representation method of the underlying working space significantly influences the efficiency of spatial reasoning algorithms of the intelligent agents. Therefore, a sparse and well-structured spatial representation is needed for these agents.

There are many ways to represent the spatial environment, such as regular and irregular grids [1], waypoint graphs [2], probabilistic roadmaps [3], and GVDs [4]. GVDs can represent the sparse skeleton of the entire spatial environment, which reduces the search space and improves the efficiency of path planning algorithms. In addition, GVDs provide the maximum clearance in regions with obstacle so that robots can avoid them. Because of these advantages, research on efficiently constructing the GVDs has recently drawn significant attention.

Triangles, squares, and hexagons are the only regular polygons that can be used to tessellate a continuous two-dimensional (2D) environment [5]. Among them, square grids are the most commonly used, and several GVDs construction algorithms based on them have been proposed, e.g., the Brushfire algorithm [6] and its improved versions (the dynamic Brushfire algorithm [7] and the method developed by Lau et al. [8]). However, the Brushfire algorithm provides no efficient mechanism for updating partial areas where local changes occur; it abandons the existing GVDs and builds a new one from scratch, making it unacceptable in a dynamic environment. To solve this problem, Kalra et al. proposed the dynamic Brushfire algorithm [7], and Lau et al. developed a novel method (referred to as the BL algorithm in this paper) [8]. Both algorithms can efficiently update GVDs when the underlying environment changes. An algorithm named dynamic topology detector (DTD) was proposed by Qin et al. [9]. In addition to generating GVDs, it extracts the connectivity among the edges and vertices of the GVDs and provides an efficient repair mechanism to dealing with local changes.

As one of the three 2D regular girds, hexagonal grids have been well studied and widely used in many fields. For

example, the hexagonal grid has long been known to be superior to the more traditional rectangular grid system in many aspects in image processing and machine vision fields [10]. In [11], a hexagonal image processing framework was proposed, advantages and disadvantages of which were also explained. In [12], a simple formula was derived for the distance between two points on a hexagonal grid, in terms of coordinates with respect to a pair of oblique axes. In [13], the binary tomography reconstruction problem of images on the hexagonal grid was studied and a (near) optimal solution was found. In addition to the applications in image processing, hexagonal grids have important applications in other fields. For example, Quijano and Garrido [14] simulated robot exploration algorithms based on hexagonal grids. They proved that these algorithms outperform those on quadrangular grids for both single and multiagent problems. Chrpa and Komenda [15] proposed a method to smooth the trajectory of a helicopter based on hexagonal grids and extended their method to multiagent pathfinding [16]. A number of computer games, e.g., Wartile and World of Warships, use hexagonal grids to represent the game world. In military simulations, the Joint Theater Level Simulation (JTLS) system of the United States also uses hexagonal grids to represent the terrain of the battlefield. Hexagonal grids have many advantages over square grids. First, when dividing the same area, hexagonal grids provide a higher spatial resolution than square grids [17]. Second, each cell in a hexagonal grid has six neighbours whose cell centroids are at the same distance [18], as seen in Figure 1. Third, hexagonal grids suffer less from orientation bias and sampling bias from edge effects since the distances to the centroids of the six adjacent cells are the same [19].

The square-grid-based GVDs generated from square-grid-based map can help to improve the efficiency of path planning tasks of intelligent agents. However, methods that employ this GVDs construction technique are not used in robotics, computer games, and military simulations, where the environments are represented with hexagonal grids. To date, there are no related studies that have focused on hexagon-based GVD generation for the path planning of intelligent agents. Pathfinding tasks on hexagonal grids are quantitatively implemented by intelligent agents in these fields, with the Morris [20], A*, and iterative deepening A* (IDA*) [21] algorithms being the most commonly used. However, their time and space complexities dramatically increase when the search area becomes larger or the number of hexagonal grids increases. In order to improve the efficiency of path planning tasks in a hexagon-based environment, the ability to construct hexagon-based GVDs is crucial for an intelligent agent that moves in a large area.

The main focus of this paper is to construct GVDs from a preexisting hexagon-based representation of an environment. An algorithm named hexagon-based crystal growth (HCG) is presented, which extracts GVDs from a hexagonal grid map. Further, two thinning patterns are proposed to obtain one-cell-wide GVDs from rough hexagon-based GVDs. Then, HCG is compared to the hexagon-based Brushfire algorithm. The usefulness of hexagon-based GVDs for the path planning tasks of agents is also demonstrated with some simulation scenarios.

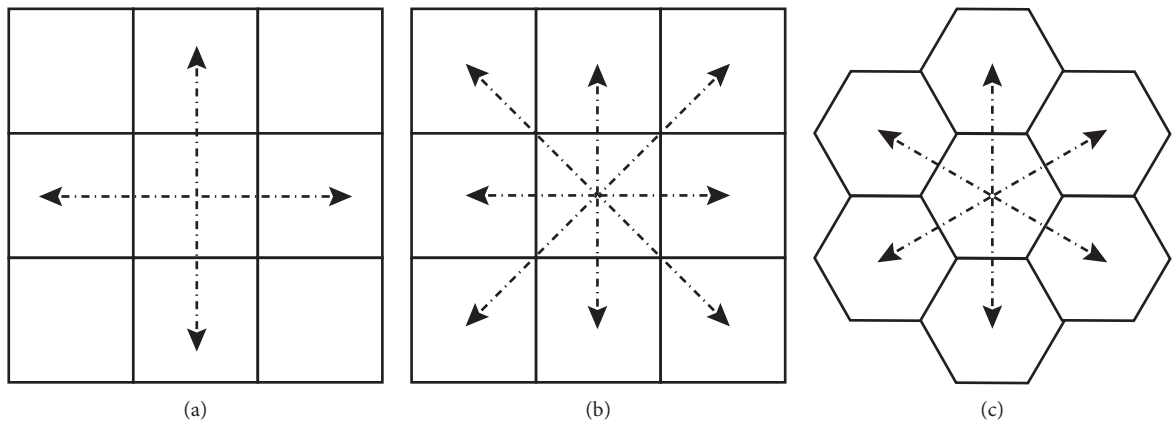
The outline of this paper is as follows: First, existing GVD construction algorithms are briefly reviewed. Among the leading algorithms for extracting GVDs from square grid maps, the Brushfire algorithm is the most fundamental. Owing to geometric differences between square and hexagonal grids, the Brushfire algorithm cannot be directly applied to hexagon-based GVD construction and needs to be modified to accommodate hexagonal grids. Next, the process of constructing a hexagonal grid occupation map is detailed. Then, the data structure employed by the presented algorithms is presented. After that, the hexagon-based Brushfire algorithm, HCG, and two thinning patterns are presented, all of which are illustrated through pseudocode. Next, HCG is compared with the hexagon-based Brushfire algorithm, and the usefulness of hexagon-based GVD metrics for path planning tasks is tested. Finally, the conclusions are presented.

2. Related Work

Voronoi diagrams, named after Georgy Voronoi, have been used to address different problems in various fields, including anthropology, archaeology, astronomy, biology, cartography, chemistry, computational geometry, geography, robotics, and planning [22]. To address the complexity of real-world problems, a number of advanced Voronoi diagrams have been developed, e.g., weighted Voronoi diagrams [23], city Voronoi diagrams [24], and GVDs [4]. In the field of robotics, GVDs have been extensively used to plan a path that stays as far away from obstacles as possible. As a reduced search space, it can help to reduce computation time significantly.

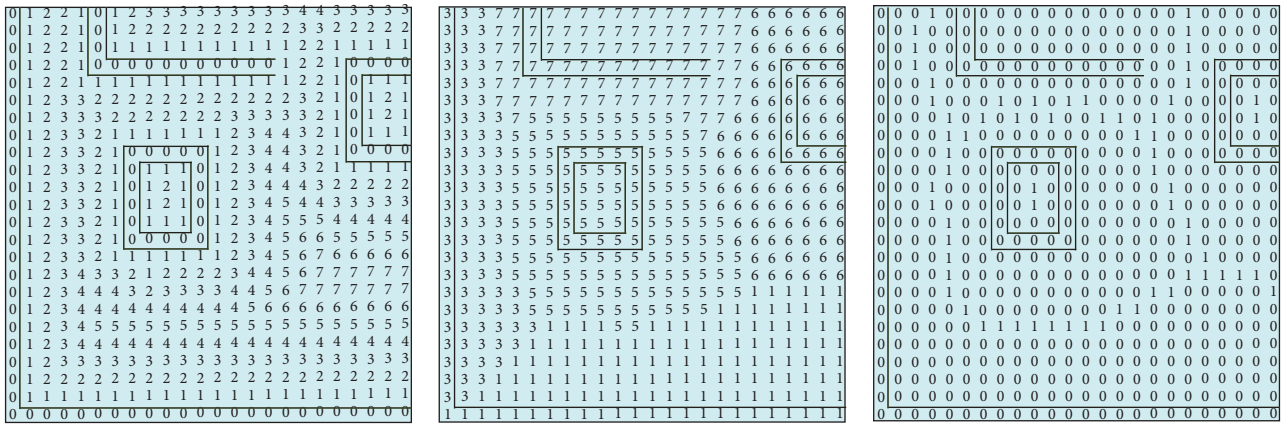
Existing algorithms for building 2D GVDs can be roughly divided into two types according to the type of input data: vector data and raster data, which are called vector- and raster-based algorithms, respectively [25]. GVDs built by vector-based algorithms are accurately and sparsely represented as a set of parametric lines or curves, which separate different sites in space [26, 27]. There are also local update mechanisms for various local changes, e.g., moving sites [28] and inserted or deleted sites [29]. Despite these advantages, vector-based algorithms are not suitable for robots whose working spaces are represented as grid maps.

Raster-based algorithms are very practical for grid maps, and they have received a significant amount of attention. As discussed in the previous section, most existing raster-based algorithms, e.g., the Brushfire, dynamic Brushfire, BL, and DTD algorithms, are based on square-grid maps. Although the performance and application scenarios of these algorithms are different, similarities in their algorithmic principles exist. They all need to generate three metric matrices ($dist_s$, $obst_s$, and $voro_s$) to represent a GVD. The matrix $dist_s$ stores the discrete or actual Euclidean distance between an arbitrary entry (denoted by s) and the site cell from which s



(a) (b) (c)

Figure 1: Square grids with (a) four and (b) eight neighbours. (c) Hexagonal grid with six neighbours.



(a) (b) (c)

Figure 2: Three GVD matrices constructed by the Brushfire, dynamic Brushfire, BL, and DTD algorithms, (a) corresponding $dist_s$ matrix, where each entry stores the integral distance to its nearest site cell, (b) corresponding $obst_s$ matrix, where each entry stores the site identifier and the exact coordinates (not represented here) of its nearest site cell, and (c) corresponding $voro_s$ matrix, where each entry shows whether the site cell belongs to the GVD (registered as 1) or not (registered as 0). This figure is cited from [9].

propagates. The matrix $obst_s$ registers the site identifier and the coordinates of the exact site cell to which s is currently the closest. Finally, the matrix $voro_s$ is a Boolean matrix that indicates whether s is a GVD cell [9]. Figure 2 shows the three matrices. Some other researchers have improved some of these algorithms. For example, Scherer et al. propagated the actual Euclidean distance instead of the grid steps used by the dynamic Brushfire algorithm from the exact source cell to greatly reduce the relative error [30]. By using the "thinning patterns" proposed in [31], Lau et al. provided additional thinning steps to obtain one-cell-wide edges, which makes the resulting GVD edges sparser.

3. Construction of a Hexagonal-Grid Occupation Map

Hexagonal grids are used to discretise the maximum active space of intelligent agents. As illustrated in Figure 3, the maximum manoeuvring range of an agent is a rectangular area, the length and width of which are denoted by L and W, respectively. The length of the edge of each hexagon is denoted by r, and the distance from the centre of a hexagon to its edge is denoted by h.

The hexagonal grid is indexed as "ColID–RowID," where "ColID' and 'RowID" are the indices of the column and row in the grid, respectively. Given the maximum ColID col and maximum RowID row, r can be calculated by

$$r = \min\left(\frac{2 \times L}{3 \times \text{col} + 1}, \frac{W}{\sqrt{3} \times \text{row}}\right). \tag{1}$$

For the cells in the even columns, the coordinates of the centre point of each hexagonal grid can be calculated by

$$\begin{cases} x = \left[\text{int}\left(\dfrac{\text{ColID}}{2}\right) \times 3 + 1\right] \times r, \\ \\ y = (2 \times \text{RowID} + 1) \times h. \end{cases} \tag{2}$$

For the cells in the odd columns, the coordinates of the centre point of each hexagonal grid can be calculated by

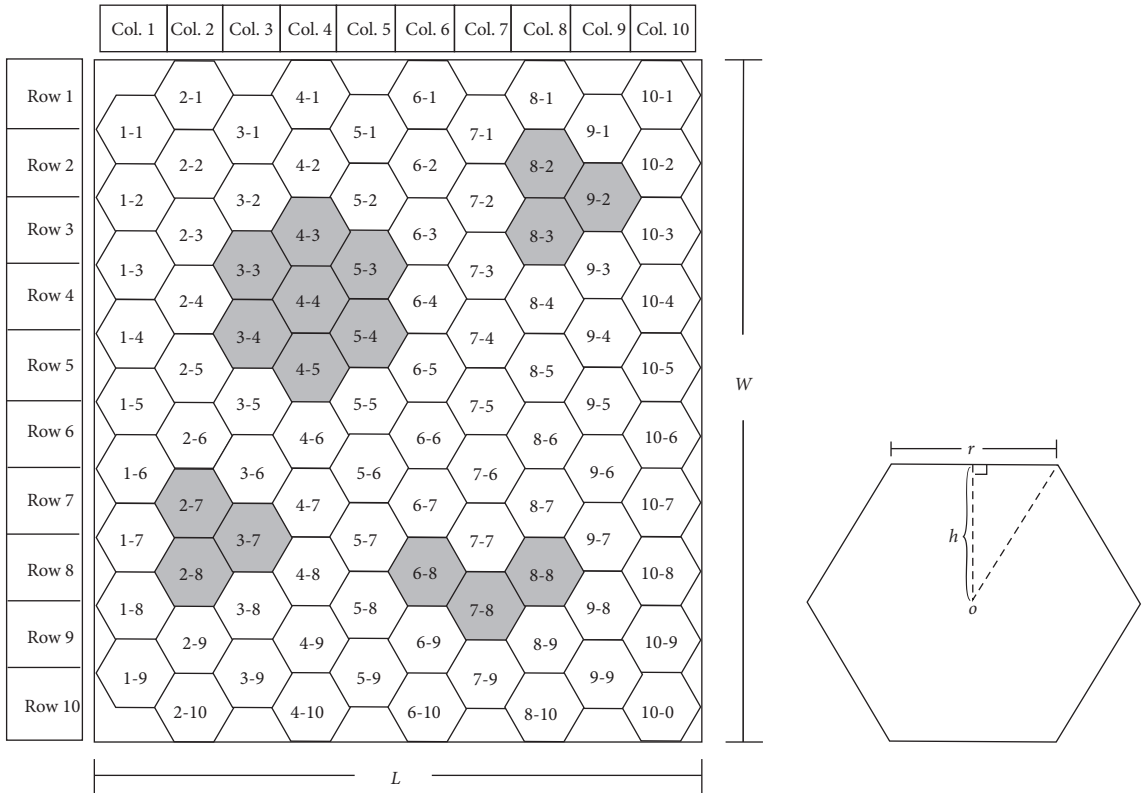


Figure 3: Example hexagonal-grid map.

$$\begin{cases} x = \left[\text{int}\left(\dfrac{\text{ColID}}{2} \right) \times 3 + 2 \right] \times r + \dfrac{r}{2}, \\ y = (2 \times \text{RowID} + 1) \times h. \end{cases} \tag{3}$$

As illustrated in Figure 3, the grey and white hexagons represent obstacles and free cells, respectively. In this paper, we use a raster-based occupation map, which represents the working space of intelligent agents, as the data source for constructing a hexagonal map. For intelligent agents, the black part in the occupation map is impassable, and the white part is passable. The main processes for hexagonal map construction are the generation of the hexagonal grid that overlies the occupation map and the calculation of the area of an obstacle that falls into each hexagon in the hexagonal grid. If the ratio of the obstacle area to the area of the hexagon is larger than a preset threshold, then the hexagon is set as an obstacle that is impassable. In contrast, if the ratio is smaller than the preset threshold, the hexagon is set as a free hexagon that is passable. In this paper, the threshold is set as 50%.

ArcGIS 10.6 (a geographic information system) is used to convert the raster-based occupation map into a hexagonal map owing to its powerful spatial analysis capabilities. Figure 4 shows the whole construction process. First, by using the ArcGIS tool "Raster to Polygon," the raster occupation map is converted into a vector map. Then, according to the range of the generated vector map, the ArcGIS 10.6 tool "Generate Tessellation" is used to generate a hexagonal grid, which is also in a vector format. Using the generated vector map and the hexagonal grid as input data, the ArcGIS 10.6 tool "Intersection" is employed to retrieve the intersections of the obstacles and the hexagonal grid. At the same time, the area attribute of each intersection can be obtained. All intersections whose areas are larger than 30% of the area of a single hexagon are selected by the ArcGIS 10.6 tool "Select by Area" and are exported as a new polygon feature layer, which is further converted into a point feature using the ArcGIS 10.6 tool "Feature to Point." Using the point feature layer and hexagonal grid as input data, the ArcGIS 10.6 tool "Spatial Joining" is employed to obtain the hexagonal grid occupation map, whose "Joint_Count" attribute indicates whether a hexagon is impassable (if the value of "Joint_Count" in the attribute table is 0) or passable (if the value of "Joint_Count" in the attribute table is 1).

4. Employed Data Structure

A data structure is employed to represent the components of GVDs, as shown in Figure 5. Hash tables are used to store the components that are extracted during the construction of GVDs to ensure efficient retrieval. Each component instance inserted into a table is assigned with a unique identifier. For example, vertices are identified by their coordinates in the hexagonal grid map, and edges are identified by the ID pairs that indicate the two sites it divides. The semantics of the related data objects and their attributes that will be quoted by the two algorithms are listed in Table 1.

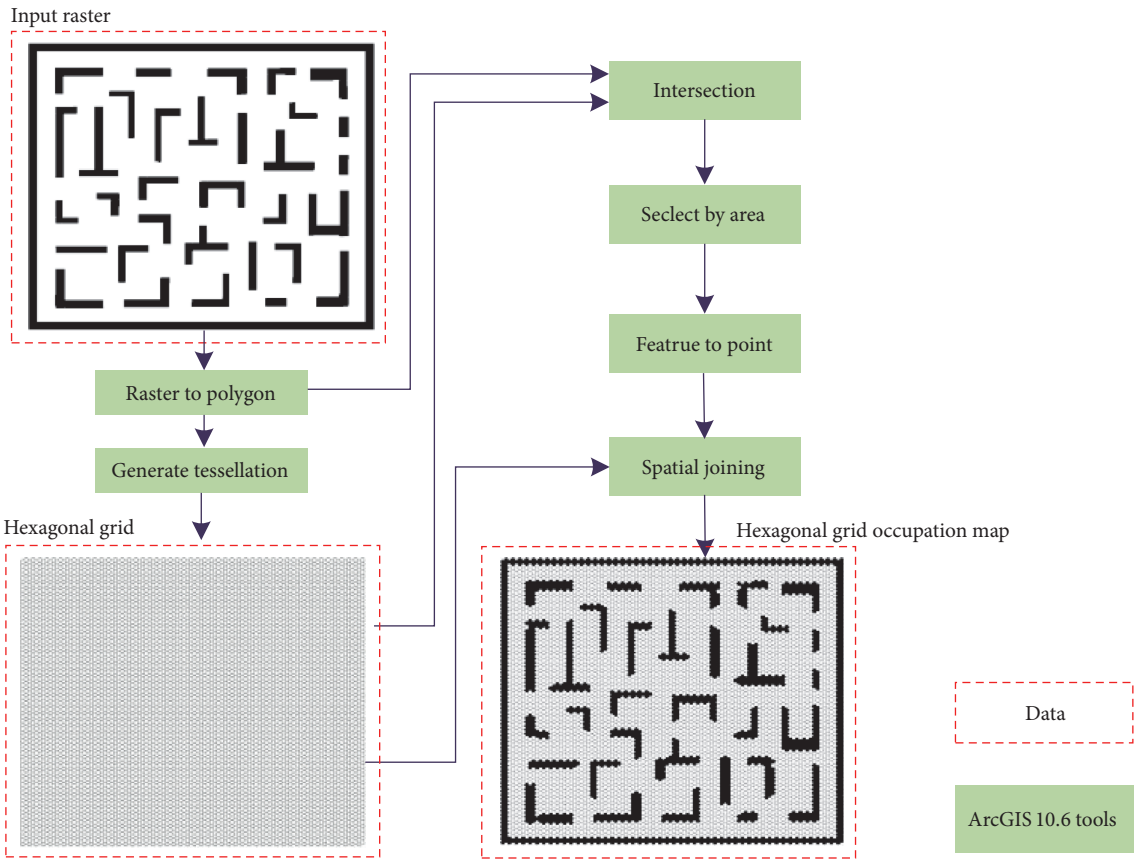


Figure 4: Process of constructing a hexagonal grid occupation map using ArcGIS 10.6.

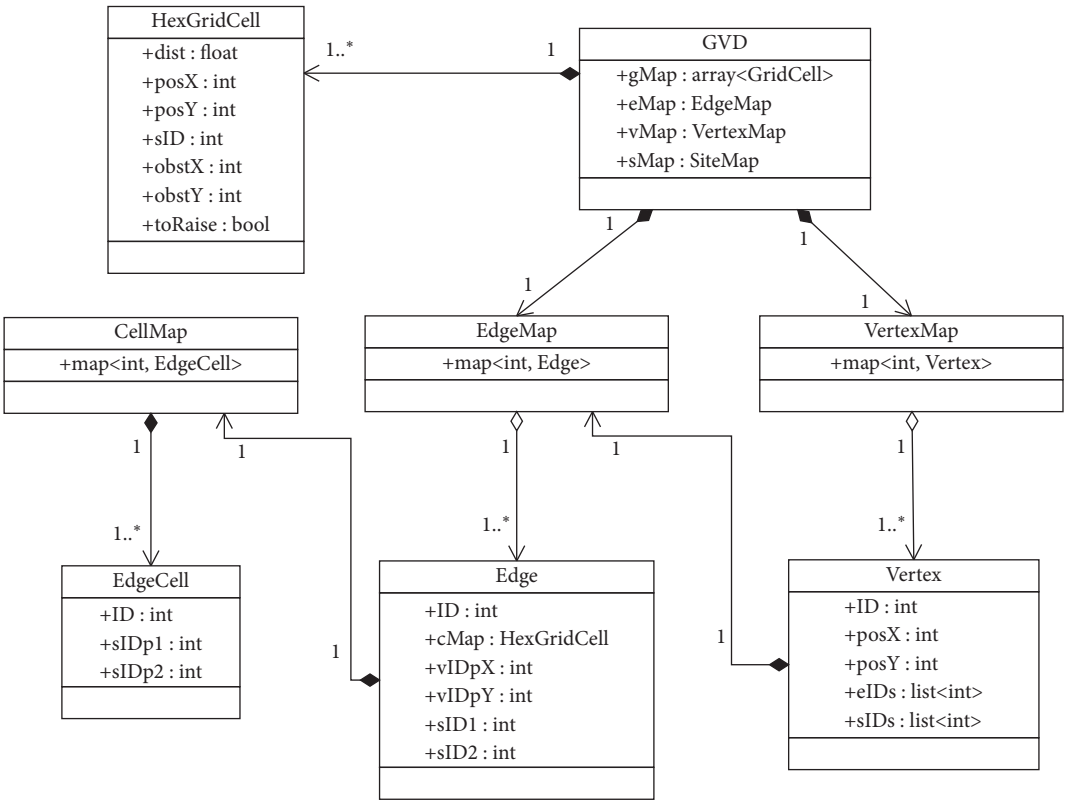


Figure 5: Class diagram for the data structure employed by DTD.

Table 1: Semantics table of the related data structure employed by DTD.

Class	Attribute	Semantics
HexGridCell	Dist:float	The Euclidean distance to the nearest site cell
	posX:int	X coordinate of the grid cell in the grid map
	posY:int	Y coordinate of the grid cell in the grid map
	voro:bool	A mark indicating if the grid cell belongs to the GVD
	obstX:int	X coordinate of the nearest site cell
	obstY:int	Y coordinate of the nearest site cell
	sID:int	Identifier of the nearest site, determined by the sequence that the site is created
	toRaise:bool	A mark indicating the propagation type of this grid (raised or lowered)
EdgeCell	ID:int	The index of the edge cell in the hash table
	sIDp1:int	Identifier indicating one of the two sites divided by this edge cell
	sIDp2:int	Identifier indicating the other site divided by this edge cell
Edge	ID:int	The index of the edge in the hash table
	cMap : HexGridCell	A hash table storing the edge cells indexed by their coordinates
	vIDp1:int	Identifier indicating one of the two vertices of the edge
	vIDp2:int	Identifier indicating the other vertex of the edge
	sIDp1:int	Identifier indicating one of the two sites divided by this edge
	sIDp2:int	Identifier indicating the other site divided by this edge
Vertex	ID:int	The index of the GVD vertex in the hash table
	posX:int	X coordinate of the GVD vertex
	posY:int	Y coordinate of the GVD vertex
	eIDs:list<int>	A list storing the IDs of the edges that are connected to the vertex
	sIDs:list<int>	A list storing the IDs of the sites that are connected to the vertex
GVD	gMap:array<HexGridCell>	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
	sMap:SiteMap	A unique hash table storing the corresponding spatial object of GVDs

5. Algorithms

In this section, on the basis of the principles and steps of the Brushfire algorithm based on a regular quadrilateral gird, this algorithm is adaptively adjusted to achieve GVD generation on a regular hexagonal grid, and a hexagon-based Brushfire algorithm is realized. After that, the HCG algorithm is proposed to generate GVDs on a hexagonal grid. A comparison between these two algorithms is also made to determine which one is more efficient.

5.1. Hexagon-Based Brushfire Algorithm. Figure 6(a) shows a flowchart of the main steps of the hexagon-based Brushfire algorithm. In Step 1, the $dist_s$ matrix, where each entry stores the integral distance to its nearest site cell, is built. In Step 2, the $obst_s$ matrix, where each entry stores the site identifier and the exact coordinates of its nearest site cell, is built. In Step 3, the $Edge_s$ matrix, where each entry shows whether the site cell belongs to the GVDs (registered as 1) or not (registered as 0), is built. The width of the generated initial rough GVD edges may be 1, 2, or more cells. In Step 4, by thinning the rough result obtained in Step 3, the one-cell-wide GVD edges are generated. Figure 6(b) shows the transitions of the GVDs in order from Step 1 to Step 4.

In the hexagon-based Brushfire algorithm in Algorithm 1, *bfQueueList* is initialised as a list of cells that are adjacent to all of the sites in the working space and is sorted in ascending order according to the distance from the cell to its nearest obstacle. That is, a higher rank is given to a cell closer to the obstacle. If *bfQueueList* is not empty, the first cell s (lines 1-2) in it is removed. $Dist_s$ is the distance from s to its nearest obstacle. If $dist_s$ is not 0 (which means that s is not a site cell), then the six cells adjacent to s are added to the list *adjCellList* (which stores the neighbouring cells of the cells in *bfQueueList*) and are sorted in ascending order of distance from each cell to its nearest obstacle (lines 3–5). On this basis, the neighbouring cell n with the smallest distance to its nearest obstacle can be taken from *adjCellList*; then, the attributes of s, including the distance to the nearest obstacle $dist_s$, the nearest obstacle $obst_s$, and the parent cell $parent_s$ can be updated according to the attributes of cell n (lines 6–9). For each cell in *adjCellList*, if it is not an obstacle cell, if its distance to the nearest obstacle is the initial value MAXDIS, and if a is not in the list *bfQueueList*, then the nearest obstacle is *Obsts* and will be added to the list *bfQueueList* (lines 10–15). After the above process, the lists $dist_s$ and $obst_s$ can be obtained.

The function **markBrushfireRoughGVDEdges**() (line 16) is used to determine whether a cell is a GVD edge cell on the basis of the site identifier of each cell. The attribute values of *bEdgeCell* for all cells are initialised as false, which means each cell in the working space is not a GVD edge cell at first. For each cell s in *allCellList*, the six cells adjacent to s are removed. For each adjacent cell n of s, if the site identifiers to which s and n belong are different and cell s is not a site cell, then the attribute of *bEdgeCell* is set to true, which means that s is a GVD edge cell (lines 17–21). Then, cell s is added to the boundary cell list. After that, it is necessary to determine the GVD edge that s belongs to and add it to the list of cells that make up the GVD edge. However, if the GVD edge to which s

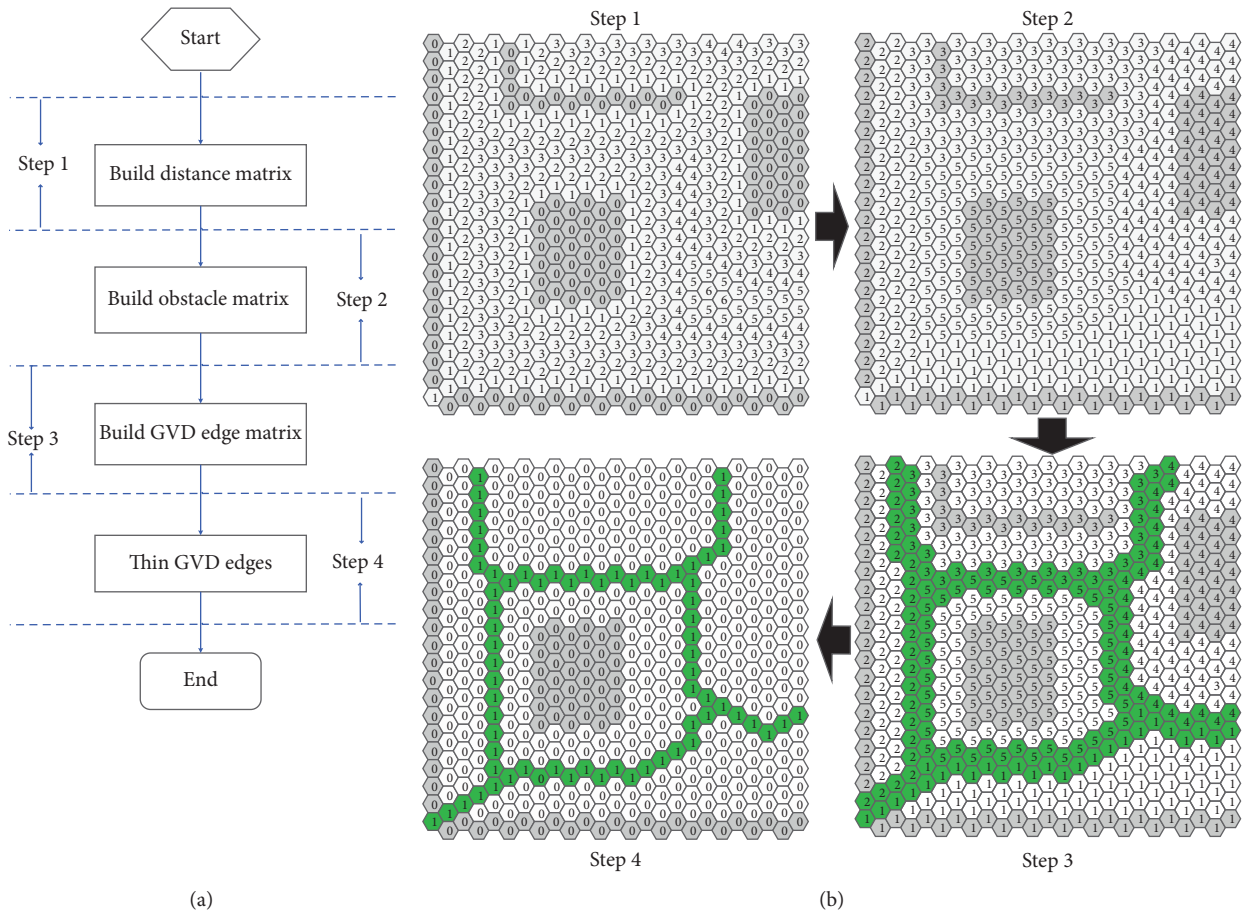


Figure 6: (a) Flowchart of the process of building a hexagon-based GVD. (b) Transitions of the hexagon-based GVDs from Step 1 to Step 4.

belongs does not yet exist, a new edge needs to be created, and s is added to the newly generated edge (lines 22–28). In order to refine the initial boundary in the future and to ensure a logically consistent single-cell-wide GVD boundary, cell s is added to the priority list *roughQueue*, and the distance from s to the nearest obstacle is increased (line 29).

5.2. Crystal-Growth Algorithm on Hexagonal Grid. The function **GenerateCrystalGrowthRoughGVDEdges**() in Algorithm 2 is used to generate the initial GVD edges, the width of which may be 1, 2, or more cells. *UnCrystalCellList* is a list of all cells except the site cells in the working space, which remain unhandled. *BoundCellList* is a list of adjacent cells of all site cells. For list *sbcl* in *boundCellList*, each cell s is removed. For each adjacent cell n of s (lines 1–4), if n is not a site cell and is not occupied, the attribute values $type_n$, $obst_n$, and $dist_n$ of n are reset (lines 5–9). Then, cell n is added to the temporary surrounding cell list *tempSBC* and is removed from *boundCellList* (lines 10–11). However, if n is not a site cell but is already occupied and if the site identifier s is different from that of n, then the type of n is set as EDGE (lines 12–14). After the above process, the list *tempSBC* is added to the new surrounding cell list *newSBCL*, and *newSBCL* is used to replace *boundCellList* (lines 15–17). At this point, a round of growth is finished, and the growth process will not stop until all of the cells in the working space are handled. The function **markCrystalGrowthRoughGVDEdges**() determines whether a cell is a GVD edge cell on the basis of the site identifier of the cells. The list *allCellList* is a list of all of the cells after the previous stage (lines 1–16). First, all of the cells in *allCellList* are traversed to check whether the type of cell s is EDGE. For all EDGE cells, their corresponding edge e is queried. If edge e does not exist, a new edge e is created, and one of the site identifiers of edge e is set equal to $obst_s$. Then, edge e is added to the edge list (lines 18–24). After that, EDGE cell s is added to edge e. The function insertQueue(*roughQueue*, s, $dist_s$) inserts s into *roughQueue* with priority $dist_s$ (lines 25–26). To refine the initial boundary in the future, cell s is added to the priority list *roughQueue*, which is sorted in ascending order by the distance from s to the nearest site.

5.3. Thinning the Rough Edges. Two thinning patterns (shown in Figure 7) are proposed and are employed by the function **pruningEdgeCell()** in Algorithm 3 to obtain one-cell-wide hexagon-based GVD edges. The input for thinning is *roughQueue*, which involves all hexagonal edge cells created by **GenerateBrushfireRoughGVDEdges**() or **GenerateCrystalGrowthRoughGVDEdges**(). All cells in

```
GenerateRoughBrushfire GVDEdges()
(1)  while bfQueueLis ≠ Ø do
(2)    s ⟵ pop(bfQueueList)
(3)    if dist_s ≠ 0 then
(4)      adjCellList ⟵ Adj6(s)
(5)      sort(adjCellList)
(6)      n ⟵ pop(adjCellList)
(7)      dist_s = dist_n + 1
(8)      obst_s = obst_n
(9)  parent_s = n
(10)       for all a ∈ adjCellList do
(11)         if type_a ≠ OBST then
(12)           if dist_a = = MAXDIS then
(13)             if a ∉ bfQueueList then
(14)               obst_a = obst_s
(15)                 insert(bfQueueList, a)
(16)  markBrushfire RoughEdge()
markBrushfire RoughGVDEdges()
(17)  for all s ∈ allCellList do
(18)    bEdgeCell ⟵ false
(19)    for all n ∈ Adj6(s) do
(20)      if obst_s ≠ obst_n ∧ type_s ≠ OBST then
(21)        bEdgeCell ⟵ true
(22)    if bEdgeCell then
(23)      e ⟵ findEdge(s)
(24)      if e = = Ø then
(25)        e ⟵ Edge()
(26)        obst_e ⟵ obst_s
(27)        insertEdge(e)
(28)      insertCell(e, s)
(29)      insertQueue(roughQueue, s, dist_s)
(30) pruningEdgeCell()
```

Algorithm 1: Pseudocode for the hexagon-based Brushfire algorithm.

roughQueue are processed in two phases. First, for each cell s in the priority queue *roughQueue*, if s matches the two thinning patterns in Figure 7, then s is retained and the next one in *roughQueue* is processed (lines 1–3). Second, if s does not satisfy the two thinning patterns in Figure 7, it is removed from the edge list to which s belongs. After all of the cells in the *roughQueue* are processed, the one-cell-wide GVDs are obtained (lines 4–6). Following the entire description of the above thinning process, the details of the two thinning patterns will be introduced.

The function **thinningPatternOne**(s) in Algorithm 3 is used to realise thinning pattern one. For each edge cell s in the *roughQueue*, if cell n, one of the six cells adjacent to s, is an edge cell, and the two cells a and b, which are both adjacent to n and s, are both unoccupied, then s will be retained in *roughQueue* (lines 7–11). This means that s belongs to the final one-cell-wide GVD edge. The function **thinningPatternTwo**(s) is used to realise thinning pattern two. For each edge cell s in the *roughQueue*, if cell n, one of six cells adjacent to s, is a not edge cell, and the two cells a and b, which are both adjacent to n and s, are both occupied (one of them is an edge cell, and the other is an edge cell or a site cell), then s will be retained in *roughQueue* (lines 12–17). This means that s belongs to the final one-cell-wide GVD edge.

6. Experiments and Analysis

In this section, statistical methods are employed to compare the hexagon-based Brushfire and HCG algorithms for some simulated scenarios. The usefulness of the hexagon-based GVDs for high-level path planning tasks is also demonstrated.

6.1. Comparison to the Hexagon-Based Brushfire Algorithm. We compared the HCG and hexagon-based Brushfire algorithms for seven scenarios, as shown in Figure 8. All scenarios are located in a static environment with a fixed size but with seven different hexagonal grid resolutions, which are 72×100, 108×150, 144×200, 180×250, 216×300, 252×350, and 288×400. In each scenario, all sites are predefined and fixed, and two algorithms were executed 10 times for each scenario. All tests were carried out with Python implementations of the algorithms running on an Intel Xeon processor.

Comparisons of the performance of the two algorithms are presented in Table 2 and Figure 9 for the computation time. From these tables and figures, it is concluded that (1) the GVD construction time of both the hexagon-based Brushfire and HCG algorithms increases in proportion to the hexagonal grid resolution for a fixed-size environment

```
GenerateCrystalGrowthRoughGVDEdges()
(1)  while unCrystalCellList ≠ Ø do
(2)    for all sbcl ∈ boundCellList do
(3)      for all s ∈ sbcl.cSiteCells do
(4)        for n ∈ Adj6(s) do
(5)    if type_n ≠ OBST then
(6)      if type_n = = EMPTY then
(7)        type_n ⟵ OCCUPY
(8)        obst_n ⟵ obst_s
(9)          dist_n ⟵ dist_s + 1
(10)         insert(tempSBC. cSiteCells, n)
(11)         unCrystalCellList.pop(n)
(12)       else if type_n = = OCCUPY then
(13)         if obst_n ≠ obst_s then
(14)         type_n ⟵ EDGE
(15)     insert(newSBCL, tempSBC)
(16)   boundCellList ⟵ newSBCL
(17)  markCrystalRoughEdge()
markCrystalGrowthRoughGVDEdges()
(18)  for all s ∈ allCellList do
(19)    if type_s = = EDGE then
(20)    e ⟵ findEdge(s)
(21)      if e = = Ø then
(22)    e ⟵ Edge()
(23)      obst_e ⟵ obst_s
(24)    insertEdge(e)
(25)    insertCell(e, s)
(26)  insertQueue(roughQueue, s, dist_s)
(27) pruningEdgeCell()
```

Algorithm 2: Pseudocode for the hexagon-based crystal-growth algorithm.

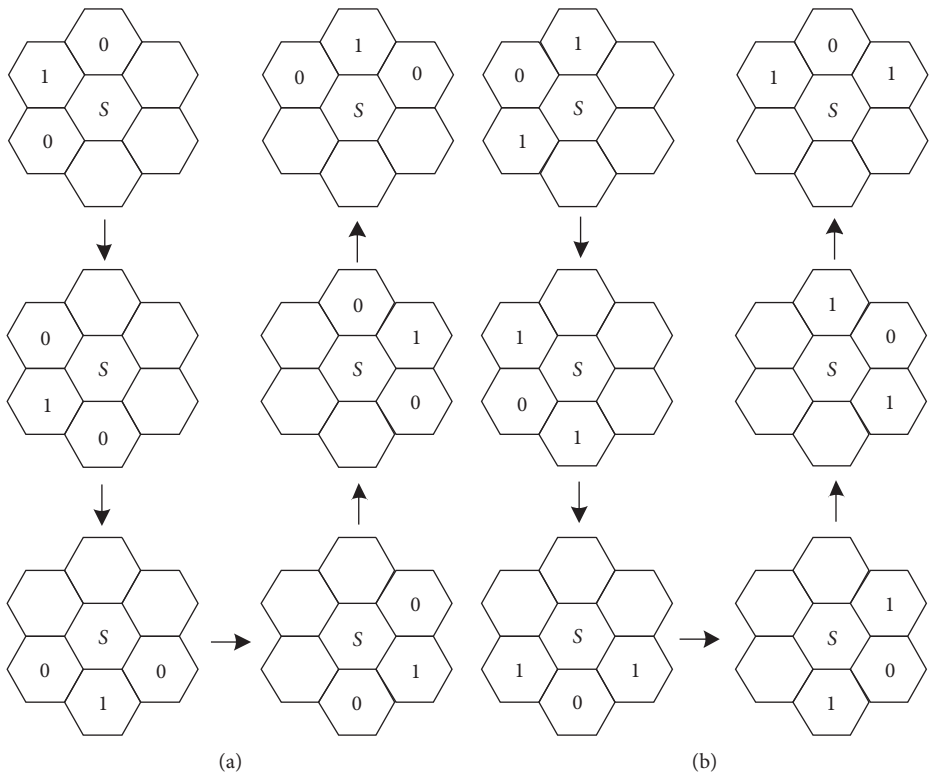


Figure 7: Patterns used by edge thinning, (a) pattern one; (b) pattern two.

```
pruningEdgeCell()
(1)   for all s ∈ roughQueue do
(2)     if fitPatternOne(s) ∨ fitPatternTwo(s) then
(3)       continue
(4)     else
(5)       e ⟵ findEdge(s)
(6)       e.remove(s)
thinningPatternOne(s)
(7)   for all n ∈ Adj6(s) do
(8)     if type_n == EDGE then
(9)       a, b ⟵ commonAdj(s, n)
(10)    if type_a == EMPTY ∧ type_b == EMPTY then
(11)      return true
thinningPatternTwo (s)
(12)  for all n ∈ Adj6(s) do
(13)    if type_n == EMPTY then
(14)      a, b ⟵ commonAdj(s, n)
(15)    if type_a == EDGE ∧
(16)      (type_b == EDGE ∨ type_b == OBST) then
(17)      return true
```

Algorithm 3: Pseudocode for obtaining one-cell-wide hexagon-based GVD edges using the two thinning patterns in Figure 7.

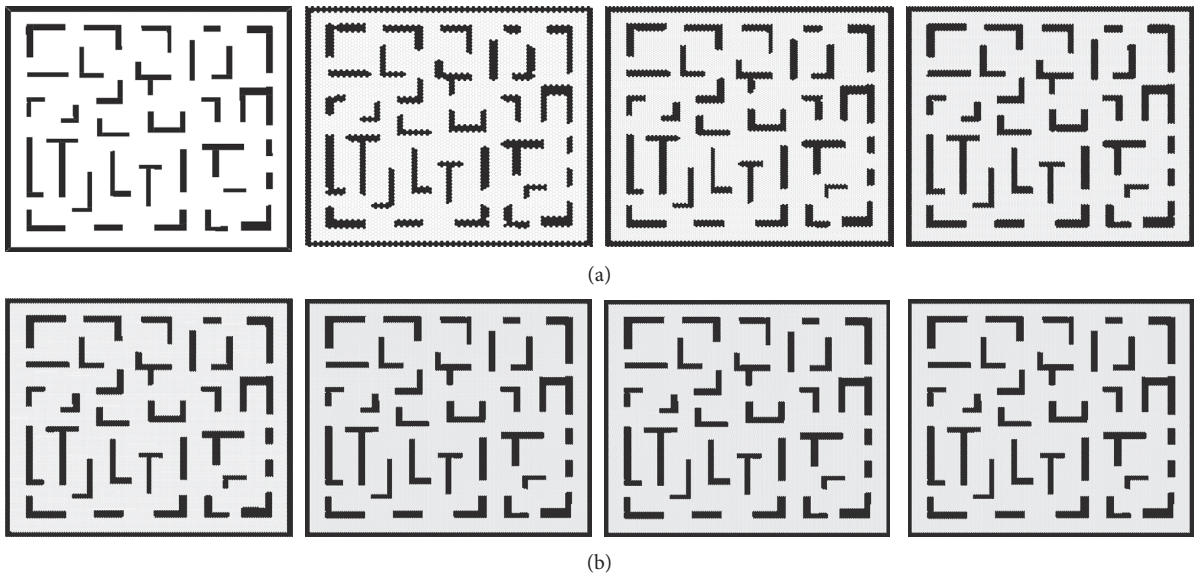


Figure 8: Original raster map and seven hexagonal grid maps with different resolutions. (a) First line, left to right: the original raster map and grid maps with resolutions of 72×100, 108×150, and 144×200. (b) Second line, left to right: grid maps with resolutions of 180×250, 216×300, 252×350, and 288×400.

Table 2: Comparison of the computation time for seven hexagon-based GVD construction scenarios. Construction from a fixed-size map with seven different resolutions.

Algorithm	Resolution						
	72×100	108×150	144×200	180×250	216×300	252×350	288×400
Hexagon-based Brushfire	7.5838	22.2965	51.6161	111.6368	182.0032	294.6933	448.1999
Hexagon-based crystal growth	1.4945	3.5701	7.3051	14.7915	23.5950	37.4815	57.4377

and (2) the HCG algorithm is much more efficient and requires less time than the hexagon-based Brushfire algorithm for each scenario with the same grid resolution.

6.2. Application Test for Path Planning. In order to demonstrate the usefulness of our algorithm for path planning tasks, seven mobile robots operating in a fixed-size map with

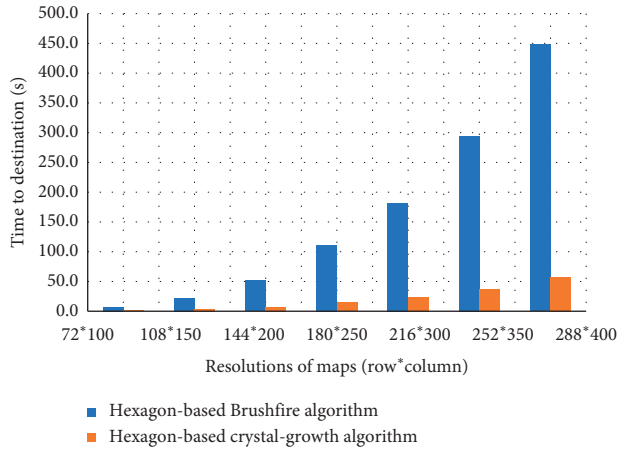


Figure 9: Comparison of the average computation time for constructing hexagon-based GVDs at different resolutions.

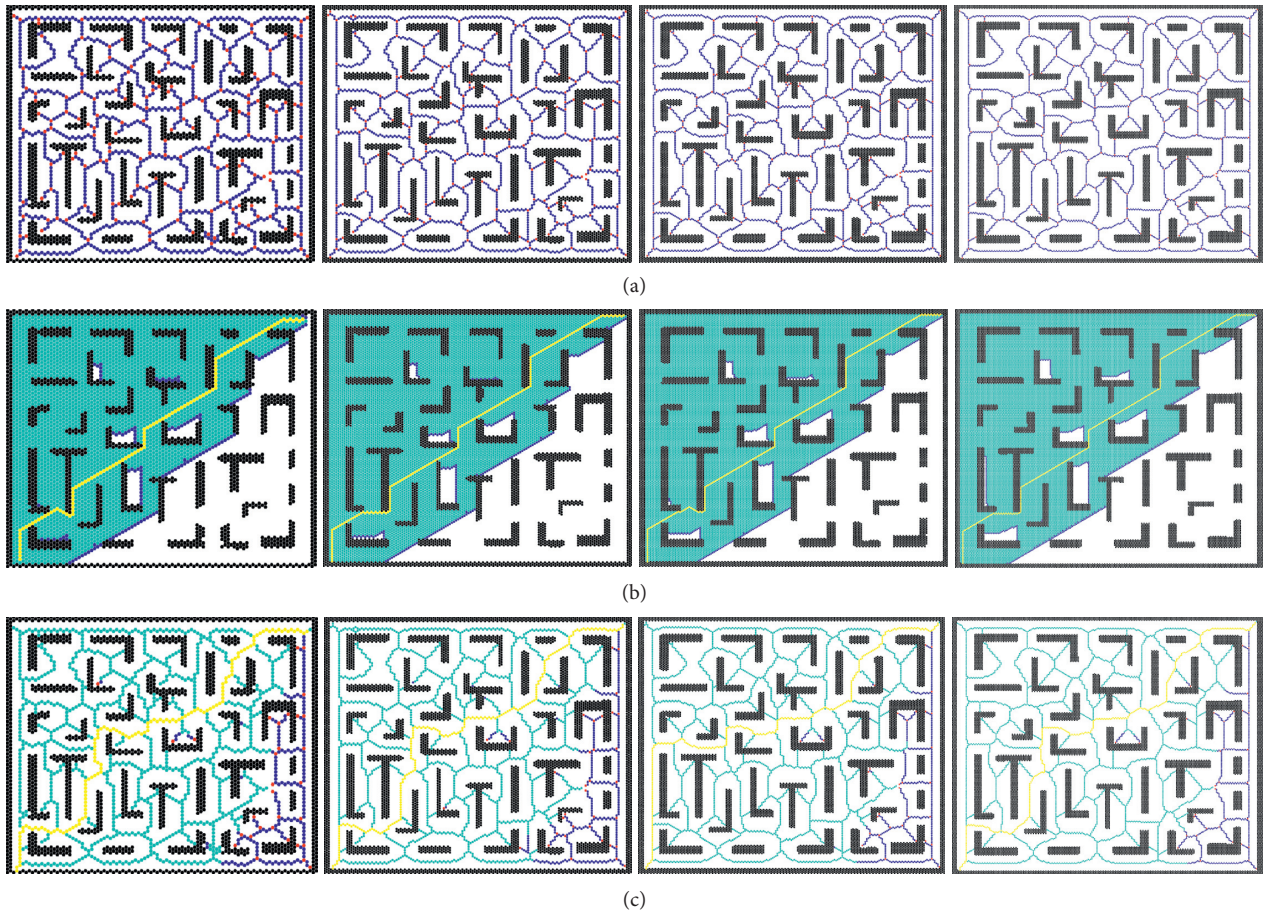


Figure 10: (a) First line, left to right: hexagon-based GVD maps with resolutions of 180×250, 216×300, 252×350, and 288×400, (b) second line, left to right: path planning results adopting A* for entire hexagonal grid maps with resolutions of 180×250, 216×300, 252×350, and 288×400, and (c) third line, left to right: path planning results adopting A* for hexagon-based GVD maps with resolutions of 180×250, 216×300, 252×350, and 288×400.

seven different resolutions (shown in Figure 10) were simulated. The starting points of the seven robots were at the same absolute coordinates at the bottom left of the map. For each search task, each robot was given a unique destination point with the same absolute coordinates at the top right of the map. However, despite the same absolute coordinates, the hexagonal grid coordinates of the seven starting points were different owing to the different resolutions. The grid coordinates of the starting and destination cells of the seven agents are listed in Table 3.

Table 3: Grid coordinates of the starting and destination cells of the seven agents.

Resolution	Start cell (row–column)	Destination cell (row–column)
72×100	(70–3)	(3–97)
108×150	(105–4)	(3–146)
144×200	(140–5)	(3–196)
180×250	(176–6)	(4–244)
216×300	(211–6)	(4–293)
252×350	(247–7)	(5–342)
288×400	(282–8)	(6–392)

Table 4: Comparison of the computation time and the average number of cell visits for path planning tasks performed by agents on both whole hexagonal grid maps and GVDs metrics with seven different resolutions.

Resolution	Whole map		GVD metrics		Ratio	
	Time 1 (s)	Cell visits 1	Time 2 (s)	Cell visits 2	Time 1/time 2	Cell visits 1/cell visits 2
72×100	5.5708	3512	0.3591	1032	15.5132	3.4031
108×150	18.1019	7463	0.6792	1550	26.6518	4.8148
144×200	54.2843	12926	1.2297	2221	44.1443	5.8199
180×250	123.3430	20560	1.7734	2753	69.5517	7.4682
216×300	236.2265	29340	2.9470	3537	80.1683	8.2951
252×350	410.7248	38662	3.6282	4008	113.2035	9.6462
288×400	684.7620	51275	4.2549	4384	160.9349	11.6959

The search space adopted by these agents was (1) the entire grid map and (2) the hexagon-based GVD metrics generated by HCG. The A* algorithm was employed by the agents to search routes. The simulation results are listed in Table 4. We can see that the agents that adopted the A* algorithm to search the entire hexagonal grid map have a higher computation time and more cell visits than those that adopted the A* algorithm to search the hexagon-based GVD metrics. The entire hexagonal grid map provides no further information about the maximum clearance to sites, making the resulting paths (in yellow in Figure 10) contain several cells near the sites, which will lead to collisions when the physical size of the agent exceeds the limited clearance. The agents that adopted the A* algorithm to search the hexagon-based GVD metrics only explore the hexagon-based GVD edge cells, significantly reducing the computation time. Each of the resulting paths (in yellow in Figure 10) consists of (1) an initial route from the starting cell to the nearest hexagon-based GVD cell, (2) a set of connecting hexagon-based GVD edges ensuring the reachability of the hexagon-based GVD departure cell, which is nearest the destination cell, and (3) a final route from the hexagon-based departure cell to the destination cell. The path planning results of the seven simulation scenarios suggest that the ratio of the computation time for searching the whole map with the A* algorithm to that searching the GVD metrics increases as the hexagonal grid resolution increases. The same result is also obtained for the ratio of the number of cells visits for searching the whole map with the A* algorithm to that searching the GVD metrics. This means that A* searching of the GVDs becomes more efficient than searching the entire hexagonal grid map as the hexagonal grid resolution increases.

7. Conclusions

In this paper, an algorithm named HCG was proposed to construct GVDs from hexagonal grid maps. Several simulation experiments were conducted to compare the HCG algorithm with the hexagon-based Brushfire algorithms (a leading grid-based GVD construction algorithm), and the results suggest that, in a hexagonal grid map with the same range and resolution, the HCG algorithm is much more efficient, requiring less time and fewer cell visits to construct hexagon-based GVDs. Moreover, two thinning patterns for obtaining one-cell-wide GVDs from rough hexagonal GVDs were proposed and were applied to both the hexagon-based Brushfire and HCG algorithms. The usefulness of the hexagonal GVD metrics in path planning was further illustrated using several representative simulation scenarios, and we found that it can significantly improve the efficiency of path planning of intelligent agents.

The proposed HCG algorithm could be applicable to the path planning of intelligent agents in fields of robotics, computer games, and military simulations, where high computing performance needs to be guaranteed. In the future, a dynamic HCG algorithm will be further explored to efficiently construct GVDs from hexagonal grid maps in which local changes may occur.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

The authors wish to thank Jian Yang for his help in preparing the manuscript. The authors would also like to thank Editage

(http://www.editage.cn) for English language editing. This research was financially supported by China's National Key R&D Program (2017YFB0503500) and National Natural Science Foundation of China (41901335).

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] D. Jia, C. Hu, K. Qin, and X. Cui, "Planar waypoint generation and path finding in dynamic environment," in *Proceedings of the 2014 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI)*, Beijing, China, October 2014.

[3] R. M. C. Santiago, A. L. De Ocampo, A. T. Ubando, A. A. Bandala, and E. P. Dadios, "Path planning for mobile robots using genetic algorithm and probabilistic roadmap," in *Proceedings of the 2017 IEEE 9th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*, pp. 1–5, Manila, Philippines, December 2017.

[4] 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.

[5] R. Klette and A. Rosenfeld, *Digital Geometry: Geometric Methods for Digital Picture Analysis*, Elsevier, 2004.

[6] 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, CA, USA, 1989.

[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] 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.

[9] L. Qin, Q. Yin, Y. Zha, and Y. Peng, "Dynamic detection of topological information from grid-based generalized Voronoi diagrams," *Mathematical Problems in Engineering*, vol. 2013, Article ID 438576, 11 pages, 2013.

[10] I. Her, "Geometric transformations on the hexagonal grid," *IEEE Transactions on Image Processing*, vol. 4, no. 9, pp. 1213–1222, 1995.

[11] L. Middleton and J. Sivaswamy, *Hexagonal Image Processing—A Practical Approach (Advances in Pattern Recognition)*, Springer, Berlin, Germany, 2005.

[12] E. Luczak and A. Rosenfeld, "Distance on a hexagonal grid," *IEEE Transactions on Computers*, vol. C-25, no. 5, pp. 532-533, 1976.

[13] T. Lukić and B. Nagy, "Regularized binary tomography on the hexagonal grid," *Physica Scripta*, vol. 94, no. 2, 2019.

[14] H. J. Quijano and L. Garrido, "Improving cooperative robot exploration using a hexagonal world representation," in *Proceedings of the Electronics, Robotics and Automotive Mechanics Conference (CERMA '07)*, pp. 450–455, Morelos, Mexico, September 2007.

[15] L. Chrpa and A. Komenda, "Smoothed hex-grid trajectory planning using helicopter dynamics," in *Proceedings of the 3rd International Conference on Agents and Artificial Intelligence (ICAART '11)*, pp. 629–632, Rome, Italy, January 2011.

[16] L. Chrpa and P. Novak, "Dynamic trajectory replanning for unmanned aircrafts supporting tactical missions in urban environments," in *Holonic and Multi-Agent Systems for Manufacturing*, V. Mařík, P. Vrba, and P. Leitão, Eds., vol. 6867 of Lecture Notes in Computer Science, pp. 256–265, Springer, Berlin, Heidelberg, 2011.

[17] R. M. Mersereau, "The processing of hexagonally sampled two-dimensional signals," *Proceedings of the IEEE*, vol. 67, no. 6, pp. 930–949, 1979.

[18] L. M. de Sousa and J. P. Leitão, "Hex-utils: a tool set supporting HexASCII hexagonal rasters," in *Proceedings of the 3rd International Conference on Geographical Information Systems Theory, Applications and Management*, pp. 177–183, Porto, Portugal, January 2017.

[19] A. Poorthuis and M. Zook, "Small stories in big data: gaining insights from large spatial point pattern datasets," *Cityscape*, vol. 17, no. 1, pp. 151–160, 2015.

[20] J. P. Morris, "An application of multi-criteria shortest path to a customizable hex-map environment," Master's thesis, Air Force Institute of Technology, Patterson, OH, USA, 2015.

[21] Y. Björnsson, M. Enzenberger, R. Holte, J. Schaeffer, and P. Yap, "Comparison of different grid abstractions for pathfinding on maps," in *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, Acapulco, Mexico, August 2003.

[22] J. Wang, M.-P. Kwan, and L. Ma, "Delimiting service area using adaptive crystal-growth Voronoi diagrams based on weighted planes: a case study in Haizhu district of Guangzhou in China," *Applied Geography*, vol. 50, pp. 108–119, 2014.

[23] F. Ricca, A. Scozzari, and B. Simeone, "Weighted Voronoi region algorithms for political districting," *Mathematical and Computer Modelling*, vol. 48, no. 9-10, pp. 1468–1477, 2008.

[24] O. Aichholzer, F. Aurenhammer, and B. N. Palop, "Quickest paths, straight skeletons, and the city Voronoi diagram," *Discrete and Computational Geometry*, vol. 31, no. 1, pp. 17–35, 2004.

[25] R. Fabbri, L. D. F. Costa, J. C. Torelli, and O. M. Bruno, "2D Euclidean distance transform algorithms: a comparative survey," *ACM Computing Surveys*, vol. 40, no. 1, pp. 1–44, 2008.

[26] 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.

[27] 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.

[28] C. M. Gold, P. R. Remmele, and T. Roos, "Voronoi methods in GIS," in *Algorithmic Foundations of Geographic Information Systems*, M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, Eds., vol. 1340 of Lecture Notes in Computer Science, pp. 21–35, Springer, Berlin, Germany, 1997.

[29] 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.

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

[31] 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.