# A Bipolar Model for Region Simplification

Yuqing Song, Jie Shen, and David Yoon

**Abstract**

In this paper we introduce an algorithm for simplifying a 2D discrete region. This algorithm is based on a bipolar model of regions. Given a discrete region, we cut its Voronoi diagram into two parts along the border of the region and each part is a tree. We use the two trees to respectively model the structures of a region and its complement, which is called the *Bipolar Model*. The simplification is done by trimming the two trees. The running time of the algorithm is $O(n \log n)$ and the space needed is $O(n)$, where $n$ is the number of points in the border of a region.

*Key words:* line simplification, bipolar model, Voronoi diagram, Voronoi tree.

## 1. Introduction

Digital lines from real applications often contain a large number of points, oscillating and angular through the points along a line. A simplification algorithm is applied to smooth out the oscillations and angles, resulting with a simplified line for a faster processing afterwards. Line simplification is especially an important problem in cartography and GIS, where digital maps consist of huge amount of lines and line simplification can greatly improve the efficiency.

The Douglas-Peucker algorithm [1], developed by David Douglas and Thomas Peucker in 1973, is one of the most commonly used line simplification algorithms. The algorithm starts by joining the two end points of a line with a straight line segment, calculating the perpendicular distance of each internal point to this line. The point with the maximum distance is identified. If the maximum distance is within a specified tolerance value, then all internal points are eliminated. Otherwise the line is divided into two segments at the maximum-distance point. This process is repeated in each of the two segments. This algorithm has a time complexity of $O(n^2)$, where $n$ is the number of vertices along an

input line. Many line simplification algorithms [6,3,4,5,6,7] have been developed by researchers from different disciplines such as cartography, mathematics, and image processing and pattern recognition.

Line simplification algorithms are classified into 5 categories by McMaster [8]: independent point algorithms, local processing routines, constrained extended local processing routines, unconstrained extended local processing routines, and global routines. The first 4 categories work sequentially by minimizing displacement. Global routines reduce points by eliminating non-salient geometric features of a line while reserving the salient ones.

A simplified line is an approximation of the original. Different criteria are used to determine the quality of an approximation. Weibel [9] specifies 4 of them: Gestalt (shape), semantic, metric, and topological constraints. Major line simplification algorithms are not tailored to meet the requirements of a specific user or task. They are based primarily on geometric error criteria, not taking into account semantic or ontological knowledge about lines [7]. In computer imagery, regions (represented as closed curves) from real images usually contain irregular angles and branches, representing the noises or texture details of objects. Thus smoothing out the irregular details is essential in capturing the main structures of a region to support effective and efficient object recognition. However, it's still an open problem how to define and identify irregular structures in a digital region.

In this paper, we propose a bipolar model to represent the structures of a planar region and then introduce a region simplification algorithm. Given a discrete region, we cut its Voronoi diagram into two parts along the border of the region and each part is a tree. We use the two trees to respectively model the structures of a region and its complement. Within each tree, we identify dominant paths. A dominant path from a given node is a path with the longest distance to a leaf node. The dominant paths are divided into dominant segments. We trim the two trees by removing the dominant segments under a given set of criteria. The two trimmed trees represent simplified regions, simplified

inward or outward, respectively. The running time of the algorithm is $O$ ($n$ log $n$) and the space needed is $O$ ($n$), where $n$ is the number of points in the border of a region.

The rest of the paper is organized as follows. In Section 2, we introduce the bipolar model for planar regions. In Section 3, we trim the Voronoi trees to simplify regions. We report the experiment results in Section 4. Section 5 concludes this paper and discusses how we may use the bipolar model for more applications.

## 2. Bipolar Model for 2D Regions

We first introduce the bipolar model to represent the structures of a planar region. Given a discrete region, we cut its Voronoi diagram into two parts along the border of the region. The inside part is a free tree and the outside part a forest. For the inside tree, we find a vertex with the maximum Euclidean distance to the boundary and make it a root. For the outside forest, we add a virtual root to represent an infinite point. The two rooted trees, inside and outside, are called *positive* and *negative Voronoi trees*, respectively. Within each tree, we identify dominant paths. A dominant path from a given node is a path with the longest distance to a leaf node. The dominant paths are divided into dominant segments.

### 2.1. Positive and Negative Voronoi Trees

In this paper we represent a region by its boundary, which is a closed and *Voronoi-continuous* curve. A closed curve, represented by a circular list of points $\{p_0, p_1, ..., p_{n-1}\}$, is called *Voronoi-continuous* if any two adjacent points ($p_i$ and $p_{(i+1)\% n}$) are Delaunay-neighbors, i.e., their Voronoi polygons share a common edge. An example is shown in Figure 1. For a closed curve, we compute its *average leap*, which is the average length of its line segments, i.e., $AverageLeap = \sum_{i=0}^{n-1} | p_i - p_{(i+1)\%n} |$. We use the average leap as a length unit to define threshold values.

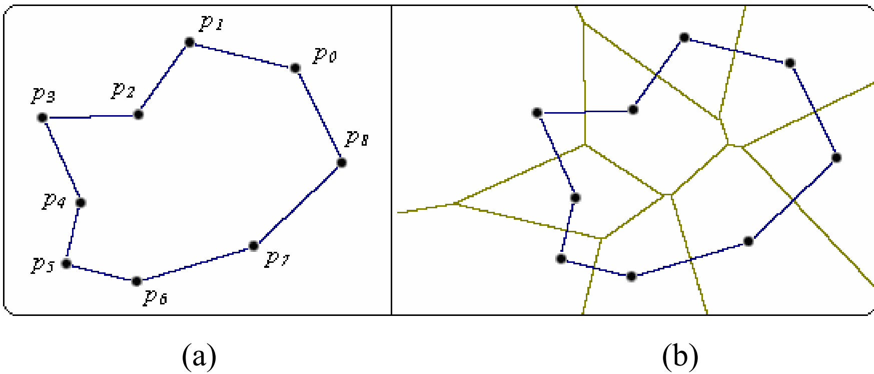
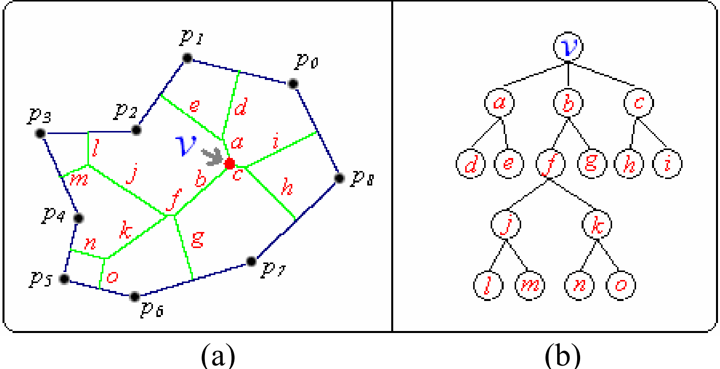



**Figure 1**: A region whose boundary is *Voronoi-continuous*. (**a**) The region. (**b**) Its Voronoi diagram.

Along the boundary of a region, its Voronoi diagram is partitioned into two parts. See Figure 1 (b).The inside part is a free tree. To make it rooted, we specify as root a vertex with the maximum Euclidean distance to the boundary, such as the vertex *v* in Figure 2 (a). The inside rooted tree is called the *positive Voronoi tree*. In representing the positive Voronoi tree, we let the root node represent the Voronoi vertex we just specified; and all other nodes represent the Voronoi edges within the region, as shown in Figure 2 (b). The outside part of a Voronoi diagram is a forest; we add a virtual root node to make it rooted. The virtual root represents an infinite point. The outside rooted tree is called the *negative Voronoi tree*. Similar to a positive Voronoi tree, for a negative Voronoi tree, all its nodes except the root represent the Voronoi edges outside of the region, as shown in Figure 2 (c) and (d).



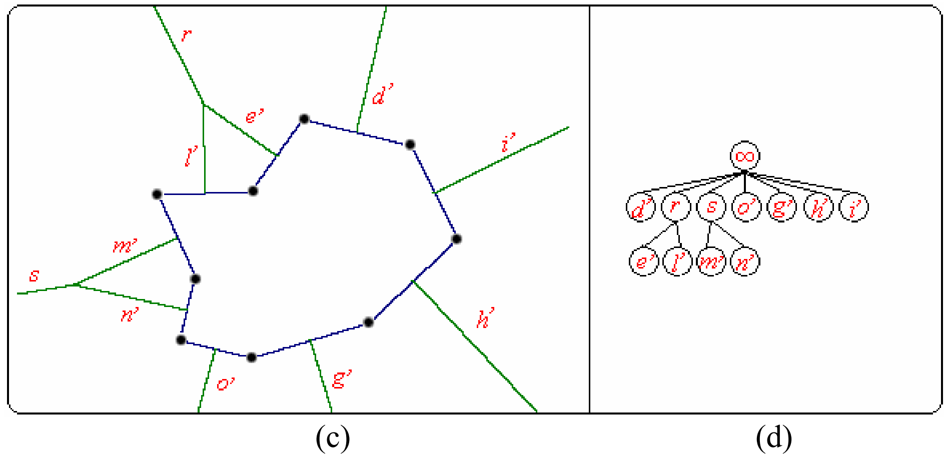



**Figure 2**: (**a**) The inside part of a Voronoi diagram. (**b**) The positive Voronoi tree. (**c**) The outside part of a Voronoi diagram. (**d**) The negative Voronoi tree.

Each non-root node of a Voronoi tree, denoted as *vNode*, has the following data items:

- The Voronoi edge it represents.
- The two input sites. For example, the node *d* in Figure 2 (a) has two input sites, $p_0$ and $p_1$.
- Its first and second child nodes.
- Its parent node.

The Voronoi edges associated with the *vNodes* are directed towards the boundary as shown in Figure 3. For each directed Voronoi edge, we record its start and end vertices. For a Voronoi edge crossing the boundary, it's cut into 2 edges, as *d* and *d'* in Figure 2. The two edges are represented by two leaf nodes in the positive and negative Voronoi trees, respectively.

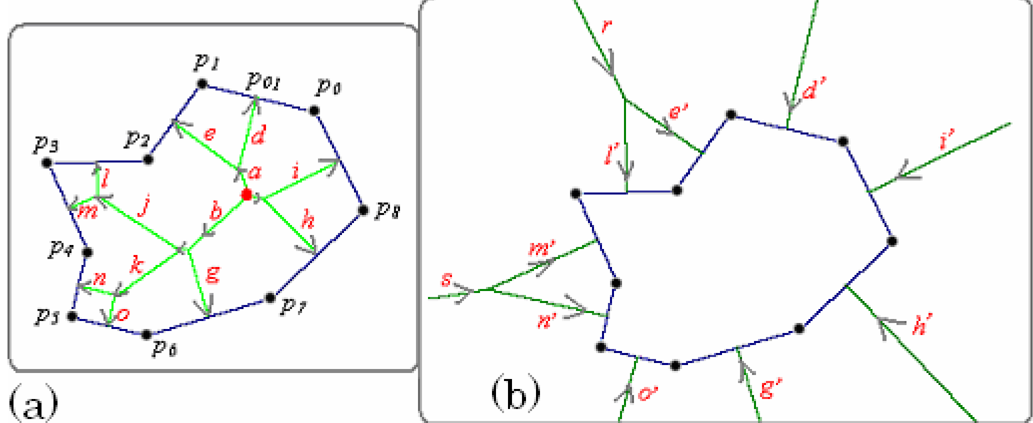


**Figure 3**: Directions of Voronoi edges in the (**a**) positive Voronoi tree and (**b**) negative Voronoi tree.

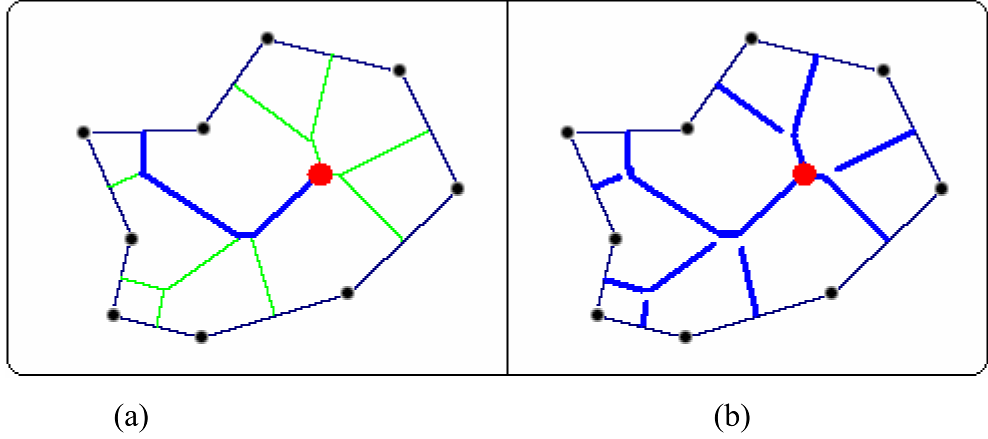
For each *vNode*, we compute the following properties:

- The *base width*, which is the Euclidean distance between the two input sites.
- The *maximum width*, which is the maximum width of this node and all its descendents.
- The *length*, which is the Euclidean distance between the two vertices of its Voronoi edge. The length of a leaf node is the length of the part (of a Voronoi edge) which belongs to this node.
- The *accumulative length*, which is the length of a longest path from this node to a leaf node. The computation of accumulative lengths begins with the leaf nodes whose accumulative lengths are equal to their lengths. For each non-leaf node, its accumulative length is equal to its length plus the greatest accumulative length of its child nodes.
- The *distance to the boundary*, which is the Euclidean distance of the start vertex of its Voronoi edge to the boundary.
- The *prominence*, which is the difference between the accumulative length and the distance to the boundary. i.e., *Prominence* = *AccumulativeLength* − *DistanceToBoundary*. A bigger prominence indicates a longer branch represented by the subtree from this node. The prominence is always non-negative. A node of zero or nearly-zero prominence indicates a circular arc, i.e., the subtree led by this node represents a circular arc in the boundary.

### *2.2.Dominant Paths and Segments*

Next we define dominant paths and segments. A *dominant path* of a vNode is a longest path from this node to a leaf node, as shown in Figure 4(a). A dominant path is computed on the basis of the accumulative lengths of vNodes. Given a vNode, the next node along its dominant path is a child with the greatest accumulative length of all child nodes of this node. Repeating this process, we find all nodes along the dominant path one by one until we reach a leaf node.

For a Voronoi tree, we compute all of its *dominant paths* recursively, as shown in Figure 4(b). For a positive Voronoi tree, starting with the root node, we find its dominant path; for its children not in the dominant path, we find their dominant paths respectively and repeat this process in their children. For a negative Voronoi tree, the computation of *dominant paths* starts with the grandchildren of the root, since the root and its child nodes have infinite accumulative lengths.



(a) (b)

**Figure 4**: (**a**) A dominant from the root (the root is shown as a big red dot inside the region). (**b**) The dominant paths computed recursively.

Once the dominant paths are computed, they are divided into *dominant segments*. The segmentation of a dominant path is done by splitting the path at the locations where the direction of the path changes greatly. A dominant path itself is a discrete curve, which is not closed. We describe a curve segmentation technique in the following paragraph. With this technique, we divide all the dominant paths into *dominant segments*.

For any non-closed curve, we compute its Voronoi diagram, which is cut into a forest by the curve, as shown in Figure 5 (b). This forest, augmented with a virtual root, called the Voronoi tree of this open curve. Similarly, we define and compute a dominant path, starting from any node of the tree, as one example shown red and thickened in Figure 5 (b). For each node in the tree, we keep track of two values, as displayed in Figure 5 (c):

- The *curve distance* between its two input sites. The curve distance of two points in a curve is the distance between the points along the curve.

- The *direction change*. Let the two input sites respectively be $p_i$ and $p_j$, and the end point of its dominant path be $q$, then the *direction change* is defined as the angle between the lines $qp_i$ and $p_jq$.

A node $n_d$ represents a significant change of the curve if its *CurveDistance* and *DirectionChange* are no less than two given threshold values, respectively. For each such node, we divide the curve at the end point of the dominant path of this node, such as the point $q$ in Figure 5 (c), which segments the curve as shown in Figure 5 (d).

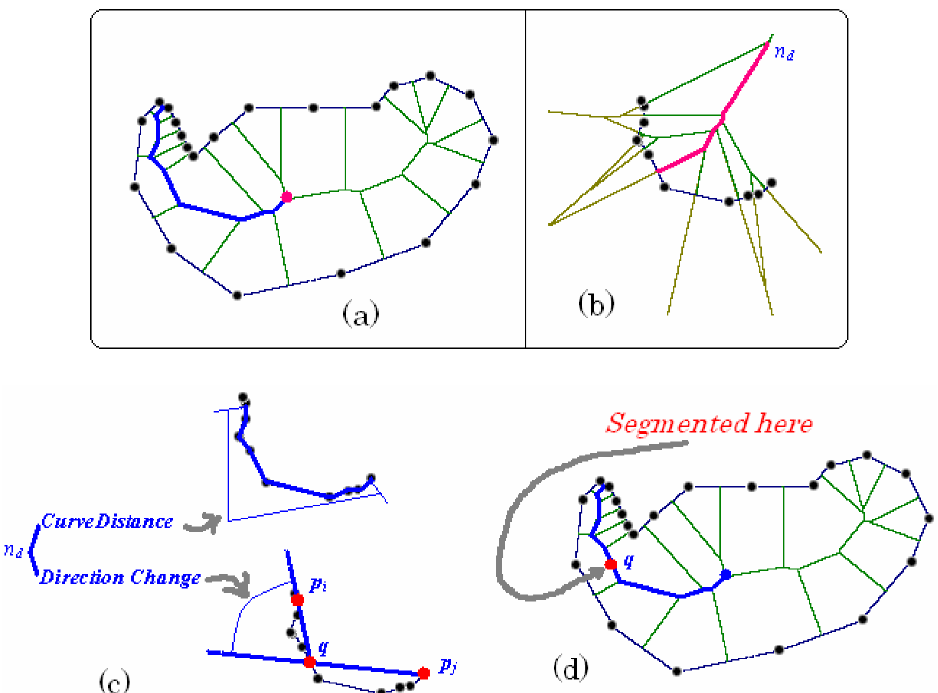



**Figure 5**: (**a**) A dominant path of a region. (**b**) The Voronoi diagram of the path. (**c**) The *CurveDistance* and *DirectionChange* of a node in the Voronoi tree of the path. (**d**) The original dominant path is segmented.

*2.3.Time and Space for Voronoi Tree Creation and Dominant Segment Computation*

In the following discussion, we assume the input region has $n$ points in its boundary.

We first investigate the space needed for Voronoi tree creation and dominant segment computation. The space is used for the following data structures:

- The positive and negative Voronoi trees of the input region.
- The dominant paths.

- The dominant segments.
- The Voronoi tree of a dominant path used for segmenting the path.

A Voronoi tree is a binary tree except that its root has more than 2 child nodes. A binary tree with $n$ leaf nodes has a size of $2n-1$. Thus a Voronoi tree of the input region has a size less than $2n-1$. For each nonleaf node, it takes constant space to represent. It takes $O(n)$ space for representing the two Voronoi trees of the input region. For the dominant paths of a Voronoi tree, as illustrated by Figure 4(b), the total number of nodes in the dominant paths is equal to the size of the tree. So the space for all dominant paths and segments is $O(n)$. The Voronoi tree of a dominant path takes $O(m)$ space, where $m$ is the number of nodes in the path and it's no more than $n$. Put all together, the total space needed is $O(n)$.

The time is used for the following operations:

- Creating the Voronoi diagram.
- Creating the Voronoi trees.
- Computing the properties of vNodes (nodes of Voronoi trees).
- Segmenting dominant paths.

For a 2D set of $n$ points, it takes $O(n \log n)$ time (that is optimal) to compute the Voronoi diagram and Delaunay triangulation (DT) [10]. Once the Voronoi diagram is constructed, we identify those edges crossing the boundary, i.e., the edges whose input sites are adjacent in the boundary. We divide each such edge into two edges. It takes $O(n)$ time. We create a vNode for each edge and then set up the parent-child relationship, taking $O(n)$ time. So the computation time for the Voronoi trees is $O(n)$. When Computing the properties of vNodes, we traverse the trees bottom-up, taking $O(n)$ time. For segmenting a dominant path of $m$ nodes, it takes $O(m \log m)$ time to construct its Voronoi diagram, $O(m)$ time to create its Voronoi tree and compute properties of the nodes, and then $O(m)$ to find the nodes representing a significant change of the path and then use these nodes to segment the path. So the time for segmenting a dominant path is $O(m \log m)$.

Summarizing the time for segmenting all dominant paths, we get $O(n \log n)$. Thus the total time for Voronoi tree creation and dominant segment computation is $O(n \log n)$.

## 3. Region Simplification by Trimming Voronoi Trees

In this section, we describe how to simplify a region by trimming the Voronoi trees.

When simplifying regions, different tasks or users have different requirements for the structures to be smoothed out. The bipolar model provides a way to allow users to formulate their requirements with the properties of the vNodes described in Section 2.1. In this paper, to test our algorithm, we choose two of the properties for the smoothing criteria: maximum width and accumulative length. Given the threshold values for the two properties, we check the start node of each dominant segment, if the two properties of the node are within the given thresholds, respectively, then we smooth out the subtree led by this node. An example is shown in Figure 6 (b).

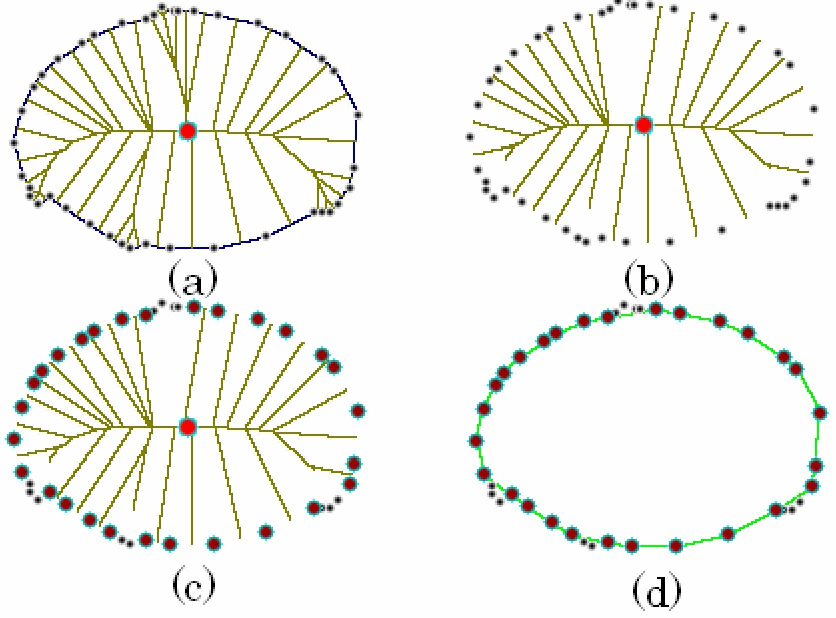



**Figure 6**: An example of inward simplification. (**a**) The input region and its positive Voronoi tree (the big dot in the center is the root vertex). (**b**) The positive Voronoi tree being trimmed. (**c**) The input sites of the leaf nodes in the trimmed tree (shown as big dots in the boundary). (**d**) The simplified boundary.

Each of the two Voronoi trees trimmed this way represents a simplified region: the positive tree represents a region simplified inward; and the negative tree represents a region simplified outward. Each trimmed tree is a connected graph of the original. For each trimmed tree, the input sites of its leaf nodes, listed in a counter-clockwise order, is a simplified boundary, representing a simplified region. See Figure 6 (d). The simplified boundary is also Voronoi-continuous, which is justified as follows. Any two adjacent points in the simplified boundary are two input sites of a leaf node of the trimmed tree. So they are Delaunay neighbors in the input region and they remain Delaunay neighbors after we simplify the region by removing some points from the boundary.

For region simplification, it takes $O(n)$ space to represent the trimmed trees and simplified regions. The time to trim the trees and identify the simplified boundary is also $O(n)$. For the whole algorithm, including constructing the Voronoi diagram, creating the Voronoi trees, computing and segmenting the dominant paths, and simplifying the region, it takes $O(n \log n)$ time and $O(n)$ space.

## 4. Experiment Results

We have implemented our algorithm in C++ and ran experiments on a Dell OptiPlex GX270 PC with 2.80GHz Pentium 4 CPU and 1GB RAM. In testing our algorithm, we use 4 threshold values for segmenting the dominant path and trimming the Voronoi trees: *T_curvedistance*, *T_directionchange*, *T_accumulativelength*, and *T_maximumwidth*. The 3 length thresholds (*T_curvedistance*, *T_accumulativelength*, and *T_maximumwidth*) use the average leap as unit. The average leap is the average length of the line segments in the boundary of the input region, as introduced in Section 2.1. As discussed in Section 2.2, a dominant path is segmented by the nodes (of the Voronoi tree of the path) whose *curve distance* and *direction change* are no less than *T_curvedistance**averageleap* and *T_directionchange*, respectively. As discussed in Section 3, a Voronoi tree is trimmed at the start nodes (of dominant segments) whose *accumulative length* and *maximum width* are no more than *T_accumulativelength**averageleap* and *T_maximumwidth*

**averageleap*, respectively. In our experiments, we set *T_curvedistance*=4.5, *T_directionchange*=0.75π, *T_accumulativelength*=4.5, and *T_maximumwidth*=4.5.

We test on regions either captured from real images or manually created. We report the results on 3 of them. Table 1 displays the input and simplified regions. Table 2 lists the numbers of points in the regions and the computation time.

| **No.** | Input Region | Simplified inward | Simplified outward |
|---|---|---|---|
| **1** | | | |
| **2** | | | |
| **3** | | | |

**Table 1:** The input and simplified regions.

| Region No. | *n* | *n*_in | *n*_out | time |
|---|---|---|---|---|
| 1 | 40 | 28 | 28 | 0.025 |
| 2 | 96 | 24 | 35 | 0.041 |

| 3 | 322 | 196 | 183 | 0.185 |
|---|---|---|---|---|

**Table 2:** Experimental results. For each input region, we list the numbers of points and the computation time:

- *n* : numbers of points in the boundary of the input region.
- *n*_in : numbers of points in a boundary simplified inward.
- *n*_out : numbers of points in a boundary simplified outward.
- time : total computation time in seconds for the two simplifications.

## 5. Conclusion and Discussion

In this paper we introduce a *bipolar model* for planar regions. Given a planar region, we cut its Voronoi diagram into two parts along the border of the region and each part is a tree. We use the two trees (called *positive* and *negative Voronoi Trees*, respectively) to model the structures of a region and its complement. We then present an algorithm for region simplification by trimming the two trees.

The bipolar model can be applied to region classification and indexing. With the trimmed Voronoi Trees, we can easily represent and compare the salient geometric features of regions and thus achieve an efficient and effective classification and indexing of 2D regions. It's out future work to discuss this application.

**References**


1. D. H. Douglas and T. K. Peucker. "*Algorithms for the reduction of the number of points required to represent a digitized line or its caricature*," The Canadian Cartographer, vol. 10, pp. 112-122, 1973.

2. R. B. McMaster. "*A Statistical Analysis of Mathematical Measures of Linear Simplification*." The American Cartographer, 13(2), pp. 103-116, 1986.

3. H. Veregin. "*Quantifying positional error induced by line simplification*." International Journal of Geographical Information Science, Vol. 14, No. 2, pp. 113-130, 2000.

4. L.J. Latecki and R. Lakamper. "Application of planar shape comparison to object retrieval in image databases. Pattern Recognition 35(1), pp. 15-29, 2002.

5. N. Shahriari and T.C. Vincent. "*Minimizing Positional Errors in Line Simplification Using Adaptive Tolerance Values*". Spatial Data Handling, pp. 153-166, ISBN 3-540-43802-5, Springer, 2002.

6. S-T. Wu and R. G. Mercedes: "*A non-self-intersection Douglas-Peucker algorithm,"* Proceedings of the Sixteenth Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI). IEEE Press, pp. 60–66, 2003.

7. L. Kulik, M. Duckham, and M. Egenhofer. "*Ontology-Driven Map Generalization,*" Journal of Visual Languages and Computing 16 (3): 245-267, 2005.

8. R. B. McMaster. "*Automatic Line Generalization*," in Cartographica, 24(2), pp. 74-111, 1987.

9. R. Weibel, "*A typology of constraints to line simplification*, " in Proceedings of 7th International Symposium on Spatial Data Handling, SDH'96, Delft, The Netherlands, 9A1--9A14, 1996.

10. M. I. Shamos and D. Hoey. "*Closest-point problems*". In Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci., pages 151-162, 1975.