Closed Loop Simplification

Release 0.00

David Doria

July 18, 2011

Army Research Laboratory, Aberdeen MD

Abstract

This document presents an implementation of an algorithm to find a low edge-count approximation of a complex, discrete, 2D closed contour. This implementation is based on the algorithm described in "Using Aerial Lidar Data to Segment And Model Buildings" and "A Bayesian Approach to Building Footprint Extraction from Aerial LIDAR Data."

The code is available here: https://github.com/daviddoria/ClosedLoopSimplification

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3302]

Contents

1 Introduction

This document presents an implementation of an algorithm to find a low edge-count approximation of a complex, discrete, 2D closed contour. Our goal is to represent the outline of an object in a simple fashion. This type of algorithm is commonly found in applications involving building detection from aerial LiDAR data.

This implementation is based on the algorithm described in [1] and [2].

2 Algorithm

2.1 Input

The input to this algorithm is an ordered set of points describing a closed contour. In Figure 1 we show a synthetic example of such a data set. The contour is visualized by joining adjacent points on the contour with an edge (a line segment).

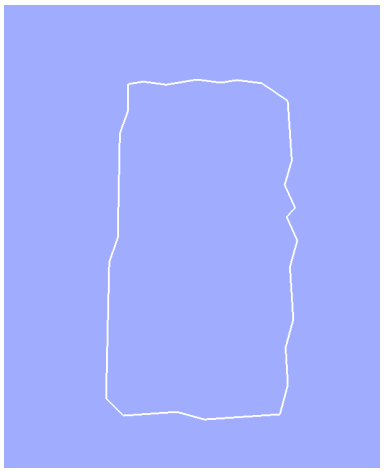


Figure 1: A closed 2D contour.

2.2 Contour Simplification

It is often desirable to reduce a contour to a simpler approximation of the same shape. For example, in the case of building detection, we often want to represent a building as a rectangle (4 line segments). A typical boundary extracted from an image can consist of hundreds of edges, mostly describing the noise in the data. To reduce the number of line segments in the boundary, we use the following procedure:

1. Create an undirected graph of the contour. At this point, we have not started towards the solution, but rather transformed the problem so that we can utilize the tools of Graph Theory.
2. Attempt to create an edge between every vertex and every other vertex. The edge is only created if the new edge passes a straightness test, described below. If all edges passed the test, the resulting graph would be a *complete graph*.

3. A straightness test is described in [1] as the sum of the distances from every point between the two end points to the proposed edge. We have found it more convenient to set this threshold based on the average of these distances, as the length of the edges then is not a factor in determining the straightness criteria. See Section 2.3 for details.

4. By finding any full loop shortest path on this graph, we will have significantly improved (i.e. reduced the line count of) our boundary. However, it is not a well posed graph theoretic problem to ask for the shortest path from a vertex to itself (a loop) and expect a non-zero answer (i.e. the shortest path from a vertex to itself is to not move at all!). To remedy this, we add the ordered vertices around the boundary twice, as well as duplicate the edges on this second loop of vertices. Now we can ask for the shortest path from vertex i to vertex $i+N$ where N is the original number of vertices in the rough boundary. Section 2.4 explains this double loop in more detail.

5. If we compute this shortest path from any random vertex, we will have a solution to our original problem of finding a low-line segment count approximation to the boundary. However, the choice of starting vertex actually can change the solution, though often only slightly. Because of this, we find the shortest path starting at all vertices, and choose the shortest one as our solution. Section 2.5 includes a demonstration of this phenomenon.

2.3 Straightness Test

Edges from the complete graph are actually created only if they approximate the points on the input contour reasonably well, to a specified threshold. Figure 2 represents the straightness calculation graphically.

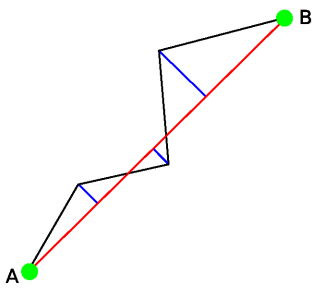
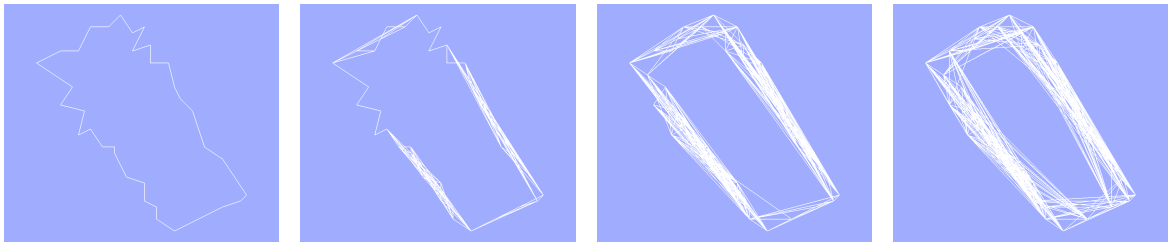


Figure 2: The straightness computation of a proposed edge between vertices A and B. Black: the original contour. Red: the proposed edge. Blue: the perpendicular line from each vertex on the contour boundary between A and B to the proposed edge. The length of these lines are averaged to obtain the "straightness" of the line.

Figure 3 shows the edges that were created using various straightness thresholds.



(a) Input contour. (b) PotentialEdges with StraightnessThreshold = 1. (c) PotentialEdges with StraightnessThreshold = 5. (d) PotentialEdges with StraightnessThreshold = 10.

Figure 3: Images of the potential edges with StraightnessThreshold = 1, 5, and 10.

2.4 “Double Graph”

Consider the graph shown in Figure 4.

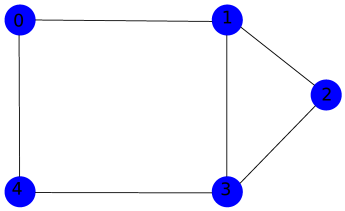


Figure 4: A simple graph.

Now consider that we want to find the shortest path “around the graph” starting at vertex 0. This question does not map to any graph theoretic terminology. However, we can construct a graph as in Figure 5 which then does have direct mapping to well known graph theory problems. We can repose the question as ““What is the shortest path from vertex 0 to vertex 5?” and obtain the path we were looking for originally. CORRECTION: There should be an edge (4,5) and NOT and edge (0,5).

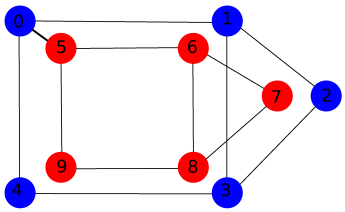
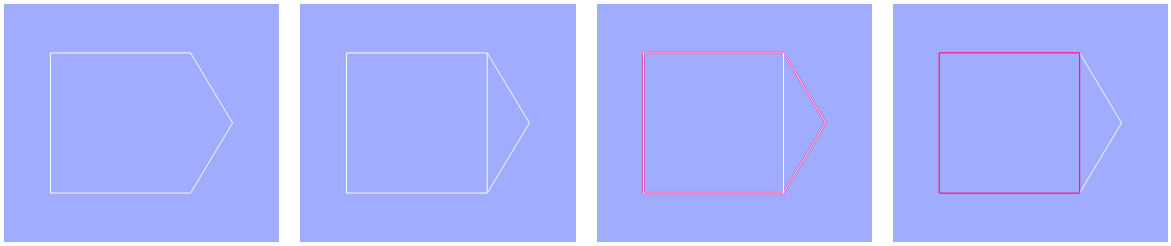


Figure 5: The blue vertices and edges are the first loop, while the second loop is shown in red.

To construct this graph, we have simply duplicated all of the vertices, while labeling them with the label $i+N$ where i the original vertex label and N is the number vertices in the original graph.

2.5 Effect of the Starting Point

To demonstrate why the shortest path must be computed from all starting points and the shortest of those paths chosen for the final output, we have contrived a set of points which clearly demonstrates this phenomenon. Figure 6 explains this graphically.

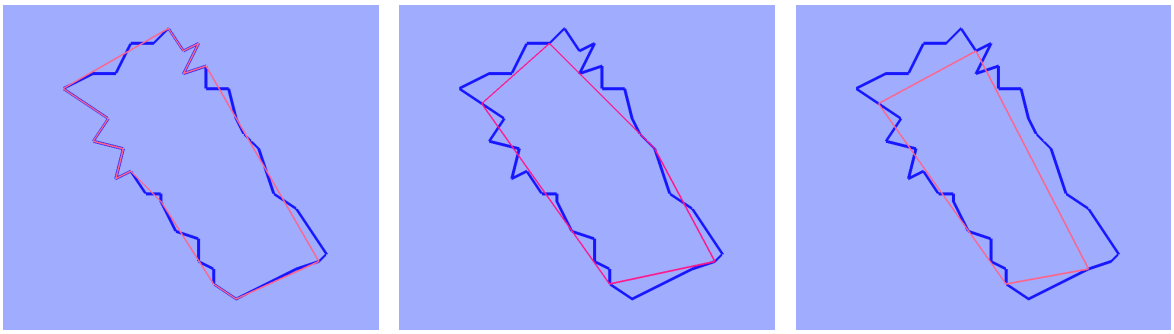


(a) Input contour. (b) Potential edges. (c) Red: the shortest path from the point not on the square. (d) Red: the shortest path from any point on the square.

Figure 6: Demonstration of why the starting point is important.

3 Demonstration

In Figure 7 we show some example simplifications of a noisy input contour. Note that the selection of the StraightnessThreshold can produce significantly varying simplifications.



(a) Simplified contour with StraightnessThreshold = 1. (b) Simplified contour with StraightnessThreshold = 5. (c) Simplified contour with StraightnessThreshold = 10.

Figure 7: A noisy input contour and its approximations at StraightnessThreshold = 1, 5, and 10. The input contour is shown in blue, while the simplifications are shown in red.

4 Code Snippet

```
vtkSmartPointer<vtkPolyData> inputContour = vtkSmartPointer<vtkPolyData>::New();
// ... Fill inputContour ...

vtkSmartPointer<vtkPolyData> simplifiedContour = vtkSmartPointer<vtkPolyData>::New();
float straightnessErrorTolerance = 1.0;
```

```
OutlineApproximation(inputContour, straightnessErrorTolerance, simplifiedContour);
```

5 Future Work

This method requires the selection of a minimum straightness parameter which has a major impact on the resulting simplification. Removing the need to manually specify this parameters would make this algorithm more robust to different data types, as well as provide the best possible results on any particular data set.

References

[1] Wang, O., *Using Aerial Lidar Data to Segment And Model Buildings*. University Of California Santa Cruz Masters Thesis, 2006 1, 3

[2] Wang. O, Lodha. S, Helmbold, D., *A Bayesian Approach to Building Footprint Extraction from Aerial LIDAR Data*. 3D Data Processing, Visualization, and Transmission 2006 1