Morphological Opening on a Graph

Release 0.00

David Doria

June 28, 2011

Army Research Laboratory, Aberdeen MD

Abstract

This document presents an implementation of an algorithm to perform a morphological opening on a graph. The intent is to remove short branches in a graph while preserving the large scale structure. This implementation is based on the algorithm described in "Efficient Closed Contour Extraction from Range Image's Edge Points". We have used the data structures from Boost Graph Library (BGL).

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

Contents

1 Introduction

This document presents an implementation of an algorithm to perform a morphological opening on a graph. This opening consists of a series of erosions, followed by a series of dilations. This is often done as a first step in contour closure algorithms. This implementation is based on the algorithm described in [1].

2 Graph Erosion

The morphological erosion operation on a graph is defined as removing all edges attached to *end points*. An end point is a vertex with only one incident edge (a vertex with degree 1). The effect of performing multiple iterations of this operation on a graph is that small branches will be ”absorbed” into a larger, parent branch.

3 Graph Dilation

The morphological dilation operation on a graph is only defined on a graph which has previously been eroded. The dilation adds back edges to current end points that were removed in a previous erosion operation.

4 Algorithm

Now that we have described the two basic operations of erosion and dilation we can describe the morphological opening operation. The algorithm proceeds as follow:

- Erode the graph multiple times until a stopping criteria is met
- Dilate the graph the same number of times as the graph was eroded

Note that this typically does not simply re-grow the original graph. If enough erosions were performed to absorb a branch completely, the branch will not grow back unless the erosion has continued so far that the root of the branch eventually becomes an end point of a future iteration.

4.1 Stopping Criteria

Naive Approach

The only parameter to be set is how many erosions to perform (as the number of dilations must be the same number).

Null Removal Difference

Recall that our goal is to preserve large structure in a graph. A good indication that we have arrived at a "stable" large structure is that the number of edges removed in successive erosion operations remains constant for some specified number of iterations. That is, if we remove 3 edges on the first erosion, then 5 edges on the next erosion (a difference of 2), the graph structure is likely still changing. However, if we move 3 edges on the first erosion, then 3 edges on the next erosion (a difference of 0), the graph structure has potentially stabilized, if we proceed with a few more erosions and we continually remove 3 edges, then we can be rather sure that the large scale structure has been found.

4.2 Speedup

Naive Approach

The most straight forward implementation is to, at each iteration, perform an exhaustive search of the vertices to determine which ones have degree 1. Since finding the end points is the only costly procedure in the algorithm, this is a bad idea.

Speedup

An exhaustive search for end points must be performed on the original graph at the beginning of the algorithm. However, at both the erosion and dilation steps, we can search a much smaller set of vertices for end points.

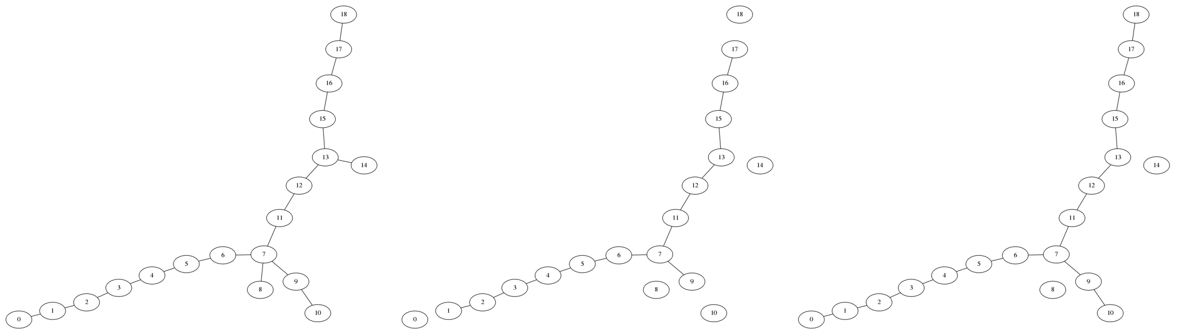
Erosion Speedup After each erosion is performed, the set of candidate vertices for the next step (either erosion or dilation) consists of the vertices that were attached to the edges that were removed, excluding the vertices that are now degree 0 (no longer attached to the graph).

Dilation Speedup After each dilation, the set of candidate vertices for the next step consists of the vertices that were newly re-attached to the graph.

If an edge was removed in the erosion process, as long as it belonged to the child-most branch that was entirely removed, it will not be regrown in the dilation process.

5 Demonstration

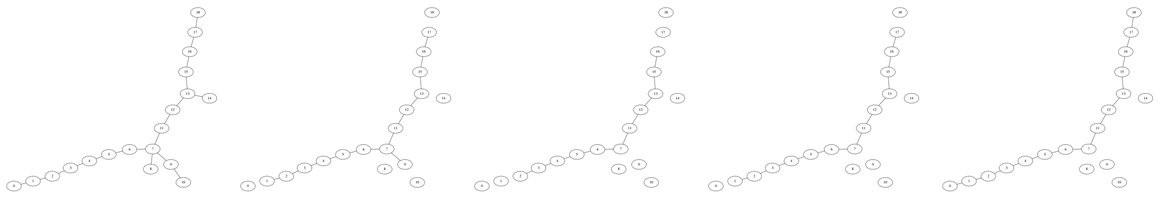
In Figure 1, we show the result of one erosion and one dilation. Note that the branches of length 1 were completely removed, while the branch of length 2 was only made shorter.



(a) Original graph. (b) After 1 erosion. (c) After 1 erosion and 1 dilation.

Figure 1: One iteration of the opening operation.

In Figure 2, we show the result of one erosion and one dilation. Note that all of the short branches were completely removed, while the large scale structure was preserved. The number of erosions and dilations are indicated in the captions. For example, $\#E = 2, \#D = 1$ indicates 2 erosions and 1 dilation have been performed.



(a) Original graph. (b) $\#E = 1, \#D = 0$. (c) $\#E = 2, \#D = 0$. (d) $\#E = 2, \#D = 1$. (e) $\#E = 2, \#D = 2$.

Figure 2: Two iterations of the opening operation.

6 Code Breakdown

- Types.h - Defines the graph structure used throughout this code.
- Helpers.h/cxx - Defines several functions that are not related to the morphological operations. These are supporting functions for the algorithm implementations.
- GraphOpeningNaive.*, GraphOpeningNaiveExample.cxx - this is the naive implementation of the algorithm, where the search for end points is exhaustive.
- GraphOpeningTracking.*, GraphOpeningTrackingExample.cxx - this is the implementation of the speedup which tracks potential end points during each operation. You should use these functions in a real application.
- Demo.cxx - There is significant overhead in producing images of graphs with missing edges without the vertices being repositioned. To do this, after each operation we create a graph with identical structure to the original graph, but with edges that have been removed marked as “invisible”. The layout program (’neato’ from ’graphviz’), then positions the vertices identically, but does not draw the edges that should have been removed. This is necessary for easy visualization of each step of the algorithm, but should not be done in a production application.

7 Code Snippet

The interface to the code is very simple, as shown below:

```
Graph graph = ReadGraph("input.dot");
unsigned int numberOfIterations = 2;
Graph openedGraph = OpenGraphFixedTracking(graph, numberOfIterations);
WriteGraph(openedGraph, "output.dot");
```

References

[1] Sappa, A, *Efficient Closed Contour Extraction from Range Image's Edge Points*. Proceedings of the 2005 IEEE International Conference on Robotics and Automation 1