An Interactive Fold-and-Cut Editor

David Benjamin and Anthony Lee

December 9, 2010

Abstract

We present an interactive fold and cut editor that creates a fold and cut pattern for a general input graph using the straight-skeleton method. Our application lets the user design their pattern interactively as the crease pattern is computed in realtime.

1 Introduction

Demaine et al. describe in [1] a general algorithm to solve the fold-and-cut problem: given a sheet of paper, perform some folds, make one cut, and try to end with a desired shape. The algorithm presented is easily run by hand and often produces very manageable results. However, it can be tedious, and some experimentation is needed to avoid complex crease patterns. We developed an interactive editor in the Java programming language to address this problem.

2 Overview

We implement a simple graphical editor "What You See Is What You Cut" editor in Java, starting from the demo editor in [2]. The editor allows for moving points and edges by dragging. Adding points and edges is done by holding the `Ctrl` key, and removing with the `Shift` key. There is very rudimentary support for grid snapping activated with the `Alt` key.

As the user manipulates the graph, we compute and update the fold-and-cut pattern. All computation is done on a separate thread, so the editor and user interface remain responsive and interactive. To implement the algorithm in [1], we use an existing straight skeleton implementation in Java by Tom Kelly, available at [2]. Kelly's code only computes straight skeletons, so we preprocess the user's graph to remove intersections and degeneracies. We then carve the problem into separate straight skeleton sub-problems for each cut region. Once the straight skeleton is computed, we link the results together and trace perpendicular creases to obtain the final crease pattern.

The editor also includes some basic features such as saving a graph, and exporting the final diagram to a PDF. For the PDF export, there exist high quality PDF libraries for Java. However, we thought it would be more fun to write our own simple one as David had already read the PDF specification, available at [3], for amusement and previously contributed a little to an open source implementation. A screenshot of our editor may be seen in figure 1.

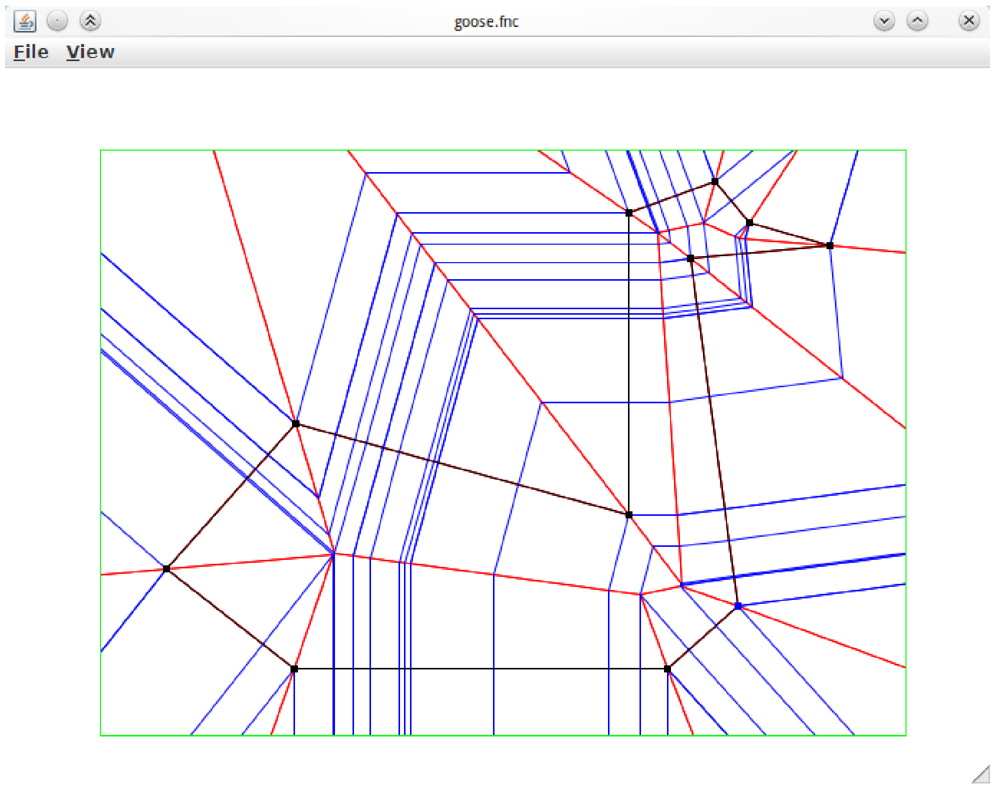

Figure 1: A sample file in our editor

3 Graph simplification

Because our program allowed the user to input a general graph, we needed to pre-process the graph considerably to convert it into a form acceptable to the straight skeleton algorithm. In the first pass, we simplified the graph; merging nearby points and cutting edges at intersecting line segments. Bentley and Ottman describe in [4] an algorithm for line segment intersection in $O((n + k) \log n)$ using a linear sweep, for n edges and k intersections. In the interest of time, we implemented a very naive algorithm instead, with a running time of $O((n + k)^2)$ for k intersections. Some degenerate cases for our intersection algorithm emerged when vertexes were located close to edges or their endpoints. For these cases, we treated them as being on the edge. Also, after the finding of intersections, we merge nearby points. Merging can also be done with a simple application of a linear sweep.

4 Straight Skeleton Input

After our graph has been simplified, we compute the dual graph and pass each face to the straight skeleton algorithm.

After computing the dual graph, we have the faces and connectivity information. (The connectivity will be used later to follow perpendicular creases.) We perform one adjustment:

the fold-and-cut algorithm requires the outer straight skeleton in addition to all the internal ones. To solve this, we create a giant square at infinity and compute a straight skeleton in there. In our implementation, infinity is set to one million. While testing, it was a paltry one thousand, so we could see it, as in figure 2. The number is large enough and zooming out is sufficiently constrained that this alone should be enough. It should not be difficult to prove bounds that guarantee the square to have no effect. (Failing that, it is easy to remove the extra faces and extend the relevant edges where necessary.)

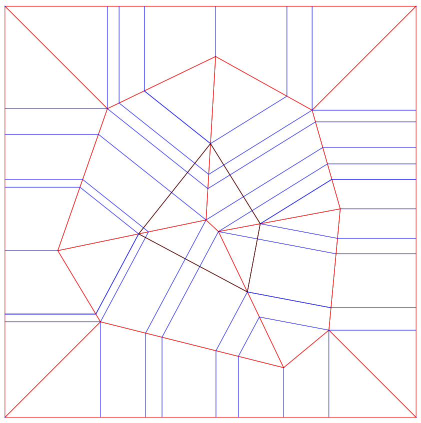

Figure 2: The square at infinity, for sufficiently small values of infinity.

While testing our code, we found that the straight skeleton implementation in [2] failed to produce a result in some degenerate edge cases. In particular, polygons which repeat a vertex along a boundary are not handled properly. Consider the fold-and-cut problem consisting of the hourglass shape in figure 3. The outer face (bounded by the square at infinity) may be represented as either containing a hole in the shape of the hourglass or two triangles. In both cases, [2] does not return a result. To solve this, we perturb every edge loop by an epsilon to start the straight-skeleton computation. Because we initially simplify the graph, this epsilon perturbation does not collide with other portions of the graph. This removes the duplicate vertices and allows our editor to handle the graph in figure 3. We maintain a mapping from the perturbed points back to the original locations to extend the straight skeleton.

This perturbation step also creates a natural place to handle generalized straight skeletons. Aichholzer and Aurenhammer generalize the straight skeleton in [5] for planar straight line graphs. We handle these additional segments by emitting two perturbed points for vertices of degree 1, places so as to be perpendicular to the segment. We also handle single points by perturbing them into small axis-aligned squares. See figure 4.

After all this preprocessing, we finally pass this data to the straight-skeleton implementation.

5 Perpendicular Creases

Once the straight skeleton is computed, we must trace the perpendicular creases, as defined in [1].

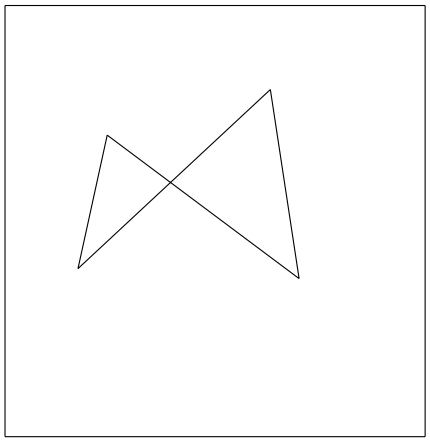

Figure 3: The problematic hourglass case.

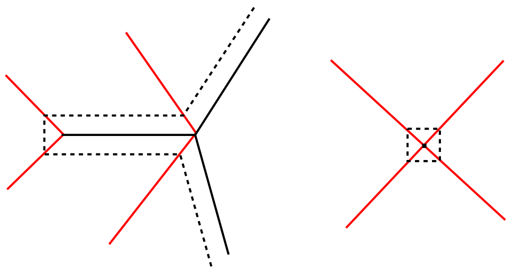

Figure 4: Perturbed generalized graphs.

By lemma 4 of [6], the polygon is monotone in the direction of its defining edge. This allows for an efficient perpendicular casting algorithm. We rotate the polygon so that its defining edge lies parallel to the x-axis. (If there are multiple defining edges, they must be colinear and thus are compatible as far as perpendicular casting is concerned.) We call this the *canonical* orientation. Because the polygon is monotone, we may cut it at the left-most and right-most points and obtain a top chain of edges and bottom chain. A perpendicular (the y-axis) intersects each of these chains, and the edges within a chain are monotonic in the x direction. The exact point of intersection may thus be found with a binary search to determine the edge and then a linear interpolation along that edge. See figure 5. One subtlety: we compute the canonical orientation from the perturbed vertex locations, but elsewhere we use the perturbed ones. This is to avoid a division by zero in the case of vertices of degree less than two.

Obtaining the perpendiculars themselves is then conceptually simple. We iterate over every vertex of every face of every skeleton output and begin following the perpendiculars from that vertex. We terminate when we reach infinity or hit a vertex; the remainder of that perpendicular will be traced at a later iteration. As noted in [1], a single perpendicular may have unboundedly or even infinitely many portions, so we bound the number of segments to a constant. (In our implementation, the limit is 40.) There are $O(n)$ perpendiculars, so

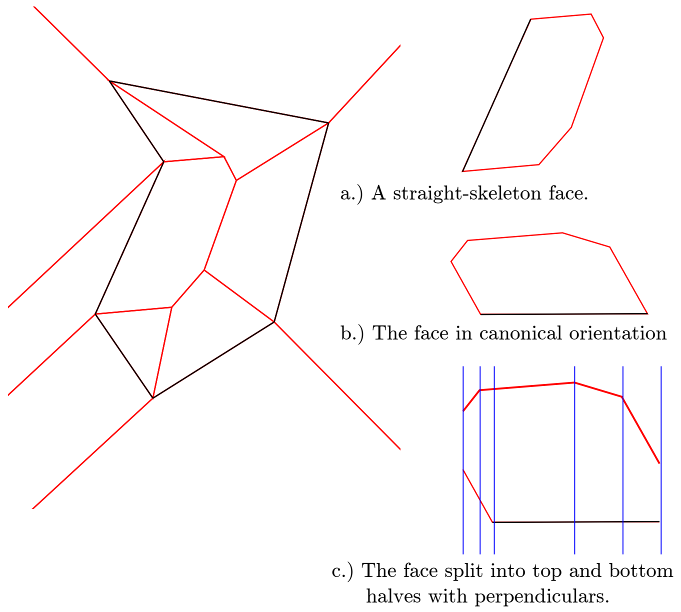
a.) A straight-skeleton face.
b.) The face in canonical orientation
c.) The face split into top and bottom halves with perpendiculars.

Figure 5: A straight-skeleton face in canonical orientation and split into top and bottom chains.

the running time of this algorithm is $O(n \log n)$.

Of course, life is rarely a matter of fluffy bunnies and strawberries when it comes to geometry algorithms, so, it cannot be this simple. At each iteration, we must find the straight skeleton face on the other side of the hit edge. For intersections that stay within the same straight skeleton, this task is fairly simple; [2]s output contains `SharedEdge` structures which link adjacent `Faces`. However, cut edges are more difficult from an implementation standpoint, though not algorithmically. While simplifying and cutting up our graph, we maintain various mappings to convert to and from simplified structures and then link together adjacent faces in the dual graph.

Finally, there is one degenerate case to handle. If a straight skeleton problem has three colinear points, the center point is often deleted. For instance, see figure 6. Skeleton face $ABCDE$ of the triangle is instead represented as ABE. As a result, the common edge of $ABCDE$ and CDF is inconsistent. We detect this case and handle such edges by hunting for information in the straight skeleton output. Our current implementation is linear in the number of colinear points per intersection, but a binary search as above is not difficult. As a result, the spiral sample in [1] is displayed correctly. The file may be found in `samples/spiral.fnc` of the project. There are two final little annoying details in our implementation. First, `SharedEdge` switches its start and end points which was the source of many confusing bugs. Second, we stop all perpendiculars as soon as they reach the bounds of the paper.

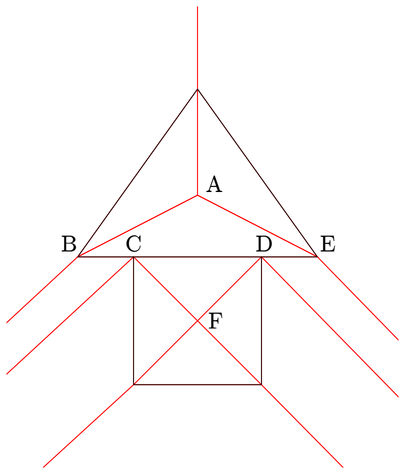

Figure 6: Degenerate case.

6 Future Work

Although the interactive fold and cut editor is already usable, there are many improvements which could be made. In terms of performance, our implementations of graph simplification and dual graph computation are very naive. Line segment intersection may be computed far efficiently than they are now in $O((n+k)\log n)$ time with a linear sweep, as demonstrated by [4]. Likewise, we could rewrite the dual graph portion to be more efficient. It should be possible to achieve sub-quadratic time also with a linear sweep and appropriate binary search tree.

Furthermore, we could improve the computed crease pattern. For now, the crease pattern does not include mountain-valley assignments. An algorithm to find them is described in [1], although it may require some modifications to the straight skeleton library to implement. In addition, while the crease patterns we generate are sufficient, they contain more creases than necessary. If possible, eliminating unused creases in the final pattern would simplify output.

More drastic simplifications to the resulting crease pattern may be obtained by forcing degeneracies in the straight skeleton and perpendiculars. The perpendiculars, in particular, tend to explode unpredictably. Degeneracies help avoid this: perpendiculars hitting vertices, straight skeleton events occurring simultaneously, etc. One way to create some degeneracies is to place vertices on a grid. Our editor inherits a very simple rectangular grid from [2], but it could be improved, for instance, with a hexagonal grid. Another simple modification could be to increase the tolerance in the straight skeleton implementation, to sacrifice accuracy for manageability. One could also imagine more involved editing features to directly merge two nearby points, for instances where the behavior of those points are well-understood.

Finally, the editor itself could be improved. Most of our effort was spent on implementing the algorithm itself. Creating loops, the primary use case of our tool, could be made simpler. Furthermore, other standard editing features could be considered. Many patterns are symmetric, so mirroring features may be useful; a symmetric graph which is slightly off often results in perpendiculars repeating wildly, whereas a perfectly symmetric graph exhibits many simplifying degeneracies.

References

[1] E. D. Demaine, M. L. Demaine, and A. Lubiw, “Folding and cutting paper,” in *Revised Papers from the Japanese Conference on Discrete and Computational Geometry*, (London, UK), pp. 104–118, Springer-Verlag, 2000.

[2] T. Kelly, “campskeleton: weighted straight skeleton implementation in java.” `http://code.google.com/p/campskeleton/`.

[3] Adobe Systems Incorporated, *Portable document format*, July 2008. `http://www.adobe.com/devnet/pdf/pdf_reference.html`.

[4] J. L. Bentley and T. A. Ottmann, “Algorithms for reporting and counting geometric intersections,” *IEEE Trans. Comput.*, vol. 28, pp. 643–647, September 1979.

[5] O. Aichholzer and F. Aurenhammer, “Straight skeletons for general polygonal figures in the plane,” in *Proceedings of the Second Annual International Conference on Computing and Combinatorics*, COCOON ’96, (London, UK), pp. 117–126, Springer-Verlag, 1996.

[6] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner, “A novel type of skeleton for polygons,” *Journal of Universal Computer Science*, vol. 1, no. 12, pp. 752–761, 1995.