Fold-and-Cut

Straight Skeleton Method Implementation

Danielle Wang

May 17, 2017

1 Introduction

I implemented the straight skeleton method for finding a solution to the fold-and-cut problem, described in [1], as a web application written in CoffeeScript. The application takes as input a set of non-intersecting polygons, (which define a unique fold-and-cut problem, since there is a canonical two-coloring of the faces) and outputs the crease pattern with mountain/valley assignments.

There is a very primitive editor which can be used to draw the desired input, or the user can manually input polygon coordinates. The crease pattern can be downloaded as a PDF or in FOLD format.

There were essentially three parts to the implementation: the straight skeleton, the perpendicular creases, and the mountain/valley assignments.

2 Straight skeleton

I computed the straight skeleton of a set of polygons, following the description in [4]. The paper describes how to compute the straight skeleton for a single oriented polygon with holes. The computation for an arbitrary set of polygons, as well as their exteriors is extremely similar. Simply replace each polygon with two oriented polygons, one in the positive orientation and one in the negative orientation.

Here's a sketch of the implementation. We keep track of a Set of Lists of Active Vertices (SLAV), which is updated as edge and split events happen. When an event happens the SLAV gets updated by removing the appropriate vertex/vertices and adding in the new skeleton vertex/vertices.

Each vertex in the SLAV points to incoming and outgoing *original* polygon edges. (No intermediate polygon edges are ever computed.) As a simple example, take the quadraliteral in Figure 1. Originally, the SLAV consists of $ABCD$, with each vertex pointing to the obvious edges. After the first edge event happens, it consists of ABE, with vertex E pointing to edges BC and DA.

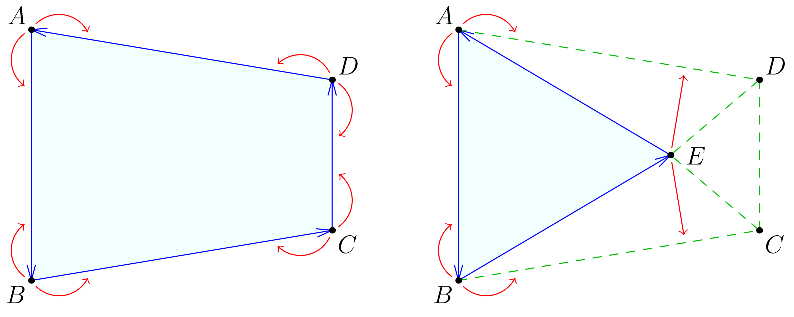

Figure 1: Example

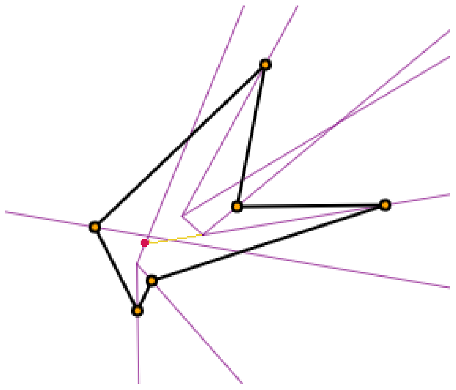

Figure 2: Need both edge events.

The paper by Felkel and Štěpán Obdržálek was extremely helpful, and I will not repeat it here, but I will note issues I ran into with the generalization and parts of the paper which were incorrect and caused me great confusion.

2.1 Edge events

The implementation involves a **priority queue** consisting of all points which could potentially be edge or split events, ordered by their distance from the polygon edge. Then we iterate over the priority queue and decide what to do with each point.

Rather importantly, when computing potential edge events, the paper says we should, for each vertex, compute the intersections of its angle bisector with the adjacent angle bisectors, and store only the nearer one. But in fact, we need to store both intersections.

For example, in Figure 2, shows what happens when only the closer intersection is stored. The red point is the closer intersection on its angle bisector, but the edge event that actually happens is the next one along.

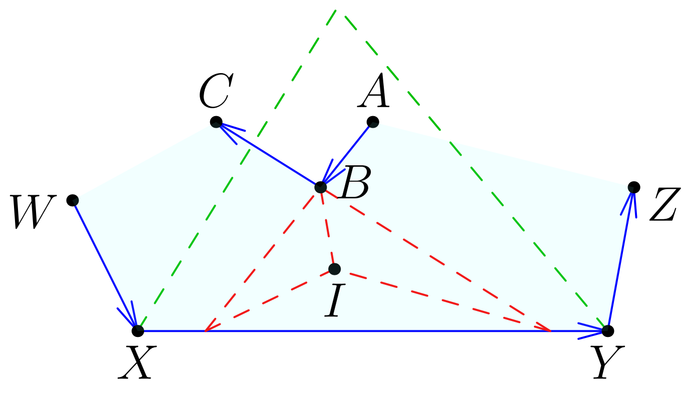

Figure 3: Strong opposite test.

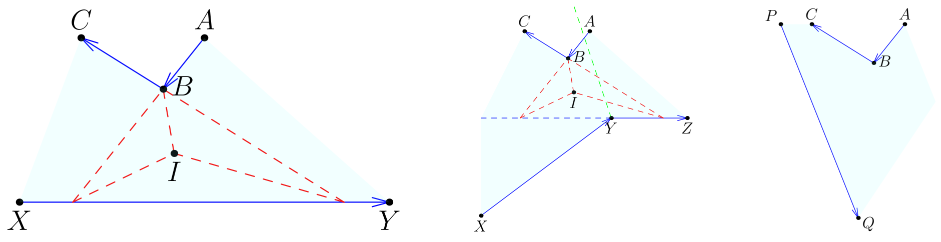

Figure 4: Weak opposite test.

2.2 Split Events

The **"opposite edge"** of a vertex is the edge which meets that vertex during a split event.

The paper states that in order to test whether an edge can be the opposite edge of a vertex, we perform the test in Figure 3, i.e. if XY is an opposite edge, the point I must lie within the green angle bisectors. However, this is incorrect. We actually also need the "weak opposite test" in Figure 4. (The edge XY passes the test as long as its angle is between the angles of edges AB and BC. Note that the figure on the right fails, while the figure in the middle passes the weak opposite test but fails the strong one.)

The correct thing to do is to store a split event into the priority queue if and only if it passes the weak opposite test, but when handling it, skip it if it fails the strong opposite test. (In addition, this test also deals with the occasional situation where a split event "turns into" an edge event, in which case we do not want to accidentally treat it is a split event.)

Also, we need to store all split events, and we also need to store both edge and split events, not just one type. Figures 5 and 6 illustrate the necessity of this. In Figure 5, the circled vertex has many potential split events which pass the weak opposite test, but only the farthest one actually becomes a split event. Also, note that the split event initially fails the strong opposite test.

In Figure 6, the closest intersection for the circled vertex is initially an edge event,

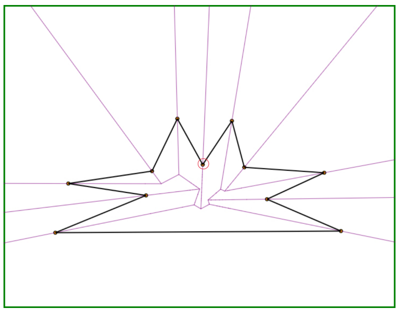

Figure 5: Need all split events.

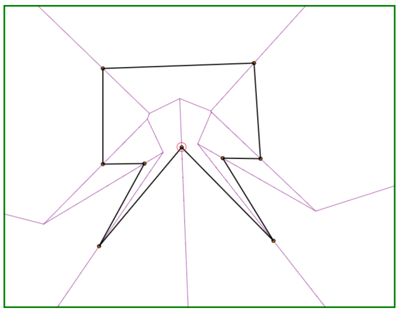

Figure 6: Need both edge events and split events.

but ultimately the circled vertex becomes a split event.

2.3 Processed vertices

A vertex becomes **"processed"** when it either merges with another during an edge event, or splits into two during a split event.

When handling an edge event I which points to vertices V_a and V_b, (meaning I is the intersection of the angle bisectors of V_a and V_b), we skip it unless *both* V_a and V_b are unprocessed. (The paper states that we should skip it only if both V_a and V_b *are* processed.)

For example, in Figure 7, the two boxed vertices are processed first (i.e. they are the first two which degenerate into one vertex). The angle bisector of the circled vertex intersects the angle bisector of the right-most boxed vertex, so it is a *potential* edge event, but that intersection point does not constitute an *actual* edge event, because when we come to it, the right-most boxed vertex has already been processed.

2.4 Leftover vertices

This is a concern only for computing the straight skeleton of negatively oriented polygons, i.e. the exteriors. After the priority queue is emptied, some vertices in the SLAV may still be unprocessed. These are the ones which go off to infinity, so we store them as

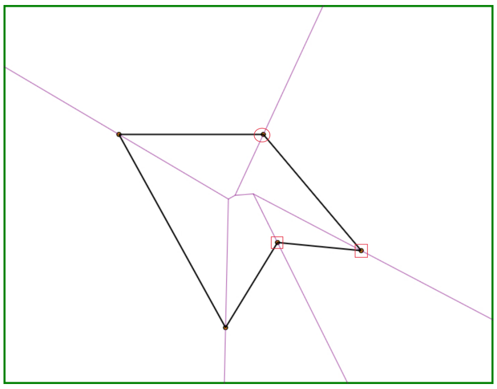
Figure 7: Need both V_a and V_b unprocessed.

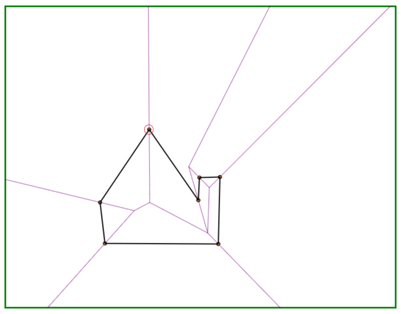
Figure 8: Leftovers.

skeleton rays.

It is *not* enough to decide whether an angle bisector becomes a skeleton ray by determining whether it intersects an adjacent angle bisector or not. For example, in Figure 8, the circled vertex is one which remains unprocessed, even though its angle bisector initially intersects an adjacent angle bisector (but that angle bisector changes direction due to an edge event).

3 Perpendiculars

The trickiest part of computing the perpendiculars is determining whether or not the perpendicular immediately leaves the face. Fortunately, it is a theorem, proven in [5], that that every face of the straight skeleton is *monotone* with respect to its corresponding cut edge, meaning that every line perpendicular to the cut edge intersects the polygon at most twice.

Thus, situations such as the one in Figure 9 are impossible, and it suffices to check whether the perpendicular intersects any of the edges of the face.

After that, I implemented a very slow way to compute the perpendiculars, by intersecting each perpendicular with every single skeleton edge/ray or cut edge, taking the

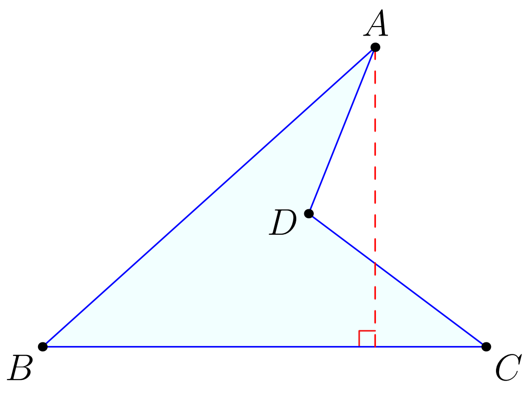

Figure 9: Leave face test.

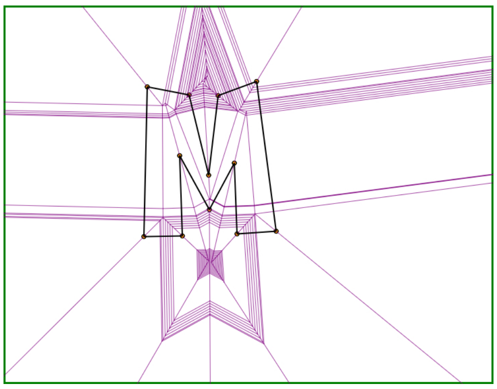

Figure 10: Example with more than 40 perpendiculars

closest intersection and reflecting the perpendicular, then repeating. In addition, every time an intersection happens, I split the skeleton edge/ray or cut edge into two edges, so the total number of edges in the resulting crease pattern is $O(p)$, and the total running time of this computation is $O(p^2)$ where p is the number of perpendiculars.

The people who implemented the fold-and-cut editor in the 2010 project describe in [3] a much faster algorithm to compute the perpendicular creases, where they actually compute all the faces and only intersect each perpendicular with the edges of the face it is traveling in.

4 Finiteness

In order to avoid infinite spiraling, the number of perpendiculars in each "chain" is limited to at most 40 perpendiculars. When there are more than 40 perpendiculars, only the crease pattern is returned, not the mountain/valley assignment.

After the perpendiculars are computed, we intersect all skeleton rays and perpendicular rays with a square (the "paper boundary") containing all of the crease pattern points.

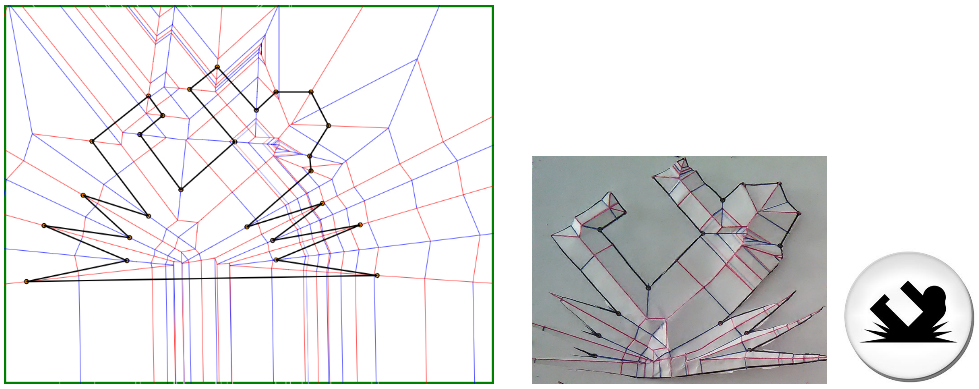

Figure 11: Example crease pattern and fold-and-cut

The function `convert` does this, converting the information into a finite `CreasePattern` object. It does not include the cut edges as part of the `CreasePattern` object. (They are just artificially drawn onto the canvas, and also not included in the FOLD file.)

5 Mountain/valley assignment

We compute the corridors by starting at a boundary edge, and following the corridor until it hits another boundary edge. The various `rotate` functions of the `CreasePattern` object help do this.

To make sure we don't repeat the same corridor twice, we only begin to follow the corridor at a boundary edge if the clock-wise most point is a the foot of a perpendicular edge (as opposed to a point where a skeleton ray hits the paper boundary, or one of the paper vertices). This ensures that one-wall corridors are only computed once each. A two-wall-corridor is uniquely determined by the two skeleton vertices which its walls originated from.

Since each corridor is folded in an accordion, it is straightforward to compute the assignments of all non-perpendicular edges in the crease pattern. The perpendicular edges' assignments are determined by rooting the shadow tree. I implemented bidirectional Dijkstra to compute distances within the `OrientedMetricTree`, from which we can read off the relative positions of the flaps.

6 Degenerate cases

The program sometimes fails to compute the straight skeleton correctly when there are parallel lines. In order to avoid this, a random number between -0.005 and 0.005 is added to each coordinate of each vertex of the input before processing.

7 Future Work

The most relevant improvement would be to extend the application to handle general planar graphs instead of just polygons. It would be mostly the same, except we would have to keep the cut edges in the crease pattern, which makes the corridor computation more complicated. The easiest way to deal with this would probably be to compute corridors ignoring cut edges first, and then insert them back into the corridors, splitting some of the corridor faces.

It would also be extremely useful to be able to force degeneracy, to simplify the resulting crease pattern. Right now the best you can do is drag around points until enough creases line up, and then you can ignore them when actually folding it.

The editor could also be significantly improved. Currently, the you can change polygon vertices once they have been drawn, but not add or remove vertices. Hitting "Esc" aborts the current polygon being drawn, and "x" deletes the previous polygon, but the polygons cannot even be deleted out of order.

The user can manually input polygon vertices, and can remember their design by saving the polygon vertices which are displayed on the page. It can also be copy-and-pasted into the manual input, but this is rather unelegant.

Another possibility is to visualize the folded state. The function `fold` I wrote attemtps to compute the folded state, but it only works under the assumption that every corridor contains a cut edge, and I gave up on fixing it in order to preserve my sanity.

Finally, the FOLD file generated often contains vertices with very large coordinates, and it might be helpful to truncate the crease pattern. In particular, sometimes when the crease pattern is put into FOLD Viewer, not much can be seen. Also, the cut edges are not included in the FOLD file, and when the 40 perpendicular limit is hit, no FOLD file is generated.

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] Erik D. Demaine and Joseph O’Rourke, *Geometric Folding Algorithms: Linkages, Origami, Polyhedra*, (New York, USA), Cambridge University Press, 2008.

[3] David Benjamin and Anthony Lee, “An Interactive Fold-and-Cut Editor,” 2010.

[4] Petr Felkel and Štěpán Obdržálek. “Straight Skeleton Implementation.” Reprinted from: Lázló Szirmay-Kalos (ed.). Proceedings of Spring Conference on Computer Graphics, Budmerice, Slo- vakia. ISBN 80-223-0837-4. Pages 210-218. `http://www.cgg.cvut.cz/Publications/felkel sccg 98.ps.gz`

[5] 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.