# THE SOURCE POSET AND UNFOLDING OF CONVEX POLYTOPES

PATRICK BYRNES

Abstract. The source poset of a convex polytope is discussed. Also, an implementation of an algorithm for finding the source poset of a 3-dimensional convex polytope is discussed.

## 1. Introduction

The source poset of a convex polytope is a partially-ordered set of points that are useful for certain computations involving the polytope. These computations include finding the Voronoi diagram of a single point on the boundary of the convex polytope and finding an "unfolding" of the polytope. Before presenting the algorithm, the Voronoi diagram of the boundary of a convex polytope and the unfolding of a convex polytope are presented.

### 1.1. The Voronoi Diagram of the Boundary of a Convex Polytope.

Let $S$ be a finite set of points in $\mathbb{R}^d$. The Voronoi diagram of $S$ is commonly viewed as the partition of $\mathbb{R}^d$ such that each cell of the partition conatins all of the points in $\mathbb{R}^d$ that are closer to some point $p \in S$ than all other points of $S$. Another way of thinking of this is that the Voronoi cell of $p \in S$ is the set of all points, $t$, in $\mathbb{R}^d$ such that the shortest path from $S$ to $t$ begins at $p$. If we consider each Voronoi cell as open (so not containing its boundary), then the union of all of the Voronoi cells is the subset of $\mathbb{R}^d$ of all points, $t$, with a unique shortest path from $S$ to $t$. Now consider $CL(S) = \mathbb{R}^d - \bigcup_{p \in S} Vor(p)$. Call this set the *cut locus* of $S$. $CL(S)$ contains all points $t$ in $\mathbb{R}^d$ such that there are multiple shortest paths from $S$ to $t$. Note that the cut locus of a set of points in $\mathbb{R}^2$ will be the edges of the Voronoi diagram of the points.

Now instead of all of $\mathbb{R}^d$, we restrict our attention to the boundary of some convex polytope $P$ in $R^d$. If $S$ is a set of points on the boundary of $P$ and $p \in S$, then we can define the Voronoi cell of $p$ on $P$ to be the set of all $t$ on the boundary of $P$ such that the unique shortest path from $S$ to $t$ starts at $p$, where we only consider paths that lie completely on the boundary of $P$. This defintion may not sound like it adds much complexity to the structure of Voronoi diagrams, but a new phenomenom is present. Now a Voronoi cell can form a border with itself. And actually, now the Voronoi diagram can have an interesting structure even if $S$ is just a single point! Consider the case of $P$ a cube in $\mathbb{R}^3$, $p$ the center of the bottom face

of the cube, $S = \{p\}$, and $t$ is the point in the center of the top of the cube. Then there are 4 distinct shortest paths from $S$ to $t$, each path consists of line segments from $p$ to the midpoint of one of the edges of the bottom face to the midpoint of the corrsponding edge of the top face to $t$. Since there is not a unique shortest path from $S$ to $t$, $t \in CL(S)$. Thus, $CL(S)$ is not empty.

1.2. **Unfolding a Convex Polytope.** The unfolding of a convex polytope, $P$, in $\mathbb{R}^d$ is a representationa of $P$ in $\mathbb{R}^{d-1}$. The intuitive idea is to make some cuts on the boundary of the polytope and then to fold the polytope along its ridges (ridges are $d-2$ dimensional faces of $P$; for the case of a convex polytope in $\mathbb{R}^3$, the ridges are the edges of the polytope). One motivation for doing this is that it maps the polytope into a smaller dimensional space which may make the structure of the polytope easier to comprehend. For instance, mapping a 4-dimensional convex polytope to some unfolding in $\mathbb{R}^3$ may greatly increase the ability to visualize the polytope. Another motivation is the possibility to construct a $d$-dimensional object by first constructing its unfolding in $\mathbb{R}^{d-1}$ and then folding the unfolding to arrive at the polytope. For instance, a cereal producer may want to put a game on the back of the box for marketing purposes. If the game requires 3-dimensional dice, then the unfolding of the dice can be printed on the 2-dimensional back of the box. These 2-dimensional printings can then be cut out and folded into 3-dimensional dice. This example is probably not terribly practical (and a bit silly), but the intent is to give the reader intuition about the ideas behind unfolding polytopes.

A prehaps more practical application is motion planning. Suppose we have a robot that is confined to the surface of some convex polytope, $P$, (for instance a representation of mountainous terrain) and we want to find a shortest path from the robot's current location, $p$, to some other location, $t$. It turns out that this can be done by cutting along the edges of the Voronoi diagram of $S = \{p\}$ on the boundary of $P$ and unfolding.

## 2. The Source Poset

Let $P$ be a convex polytope and $p$ be a point on the boundary of $P$. We will call $p$ the *source point*. Consider a light being turned on at $p$. This light will be allowed to emanate out from $p$ in a constant speed in all directions within the facet (so the light stays on the surface of $P$). When, the light reaches a ridge, the light bends and travels onto the facet on the other side of the ridge. A point $v$ is a *source image* for the facet $F$ if $v$ is in the affine span of $F$ and for some point $t$ in $F$ the light as seen from $t$ appears to be emanating from $v$. Hopefully, the following picture illustrates the situation:

## 3. The Algorithm

Given a convex polytope, $P$, and a source point, $p$, the main algorithm constructs the set of all source images for each facet of $P$. The union of all

of these sets of source images is the *source poset* of $P$ for the source point $p$, where the partial order is based on the order that the source images are created.

The idea for the construction is to start with empty sets for each facet, except for the facet, $F$, containing $p$. $F$'s source image set is initialized to be the set consisting of the single point $p$. Then, a loop is entered that at each iteration considers all of the source images that could possibly be folded across some ridge. Of all these possible source images, the one that is closest to its corresponding ridge is foldeded. The image of the source image in the folding is added to the set of source images of the corresponding facet.

For this to make sense, the idea of "could possibly be foldeded" needs to be formalized. Intuitively, the idea is that a source image can be folded across a ridge into the affine span of a facet, $F'$, if and only if there is some point $t$ in $F'$ that sees the light emanating from $p$ as emanating from the folded source image. This can actually be calculated by computing the Voronoi diagram of the set of source images in the affine span of a facet $F$ (since all of the source images for $F$ lie in a plane, this Voronoi diagram can be calculated either in the plane defining $F$ or in all of $\mathbb{R}^3$). A source image, $s$ can be folded across a ridge $R$ of $F$ if it satisfies both:

(1) The Voronoi cell of $s$ intersects the interior of $R$
(2) $s$ and $F$ are on the same side of $R$

Any such combination of source image $s$, ridge $R$, and facet $F$ is the *potential event* $(s, R, F)$. It is important to realize that a particular $(s, R, F)$ may be a potential event at some times during the running of the algorithm, but not at others. This is because the potential event is based on the Voronoi diagram of the source images in $F$, and source images will be added to $F$ thus changing the Voronoi diagram.

Now, the main loop of the algorithm can be summed up as: find the minimal event, fold the corresponding source image, update the set of all events, repeat.

However, now we need to define "minimal". First, let the *radius* of a potential event $(s, R, F)$ be the Euclidean distance from $s$ to the closest point of $R$ that is in the Voronoi cell of $s$ (where the Voronoi diagram is that of all the source images of $F$). Now, we can define the "minimal" potential event to be the potential event of the smallest radius. However, how do we choose between two potential events of the same radius? The answer is by choosing the potential event with smaller *angle*. Given a potential event $(s, R, F)$, let $w$ be the closest point of $R \cap Vor(s)$ to $s$. Then, the *angle* of $(s, R, F)$ is the angle between the vector from $w$ to $s$ and the vector in the direction of $R$ chosen so that the angle is at least $\frac{\pi}{2}$ (we could get an angle smaller than $\frac{\pi}{2}$ by choosing a vector in the opposite direction for $R$).

Why should choosing the potential event with the smaller angle be better? Well, to be honest, it doesn't make complete sense to me. However, it has something to do with the idea of the light waves from $p$ creeping up some

ridges faster than others depending on various angles. The actual proof that this works involves some differential geometry.

For completeness, here's some pseudo-code for the algorithm:

```
SourcePoset(P, p)
Input: P is a convex polytope, p is a point on the surface of P
Output: the source images in the source poset of P with source p

1.  Initialization
    - For each facet, F, of P create an empty set, SI(F)
      //SI(F) is the set of source images in the affine
      //span of F
    - Create an empty set, E,
      //E is the set of potential events
    - F' <- the facet containing p
    - Add p to SI(F')
    - For each ridge, R, contained in  F' add (p,R,F') to E

2.  while(E is not empty)
    {
      - (v,R,F) <- the minimum event in E
      - remove (v,R,F) from E
      - F' <- the facet that intersects with F to form R
      - v' <- the image of v after folding from F into F'
      - remove all events of the form (w,R',F') from E
        for any w, R'
      - add v' to SI(F')
      - recompute events of the form (w,R',F')
    }

3.  For each facet F
      - output SI(F)
```

## 4. Implementation Issues

In this section I discuss various implementation decisions I made when implementing the algorithm.

4.1. **Facets.** The facets were dealt with as follows. The geometric identity of each facet was stored as the normal vector to the facet along with a single variable that stores the value of the dot product of the normal vector and any point in the affine span of the facet. Also stored with each facet are a set of events corresponding to the facet, the set of ridges contained in the facet, and the set of source images in the affine span of the facet.

4.2. **Ridges.** Along with the geometric data describing each ridge, information encoding the fold along the ridge is also stored. Each ridge is the intersection of two facets. Because the fold along the ridge depends upon which facet is folded into the affine span of the other facet, I decided to essentially store each ridge twice. Each "half-ridge" is associated with the fold into the affine span of only one of the two facets defining the ridge. Another option would have been to store the information encoding both folds, but this would have required more care in determining which transform to use.

The geometric information stored with each ridge consists of: a point $p$ in the affine span of the ridge (so in the line containing the line segment that is the ridge), a vector $v$ in the direction of the ridge, and two real numbers $t_1$ and $t_2$. The idea is that the line through the ridge is defined as $r(t) = p + t * v$. Then, $t_1$ and $t_2$ are the values of $t$ that correspond to the two endpoints of the line segment that is the ridge.

Each ridge also had stored with it both the facets defining the ridge and the corresponding fold. The fold is represented as an afffine transform, that is as a matrix $A$ and a vector $b$ so that the fold takes a point $x$ to $Ax + b$.

4.3. **Events.** The implementation opportunities for the events are more interesting. There are two key questions. First, how do we efficiently store the events so as to make finding the minimum event easy in each iteration of the main loop? Second, how do we efficiently recompute the events after folding a source image onto a new facet?

One way to save on finding the minimum event is to store all of the events in some structure that makes this easy, say a priority queue. However, one problem is that in each iteration of the main loop many of the events could change or even be removed. But, this change is confined to the events corresponding to one facet. Thus, we can save time by building the priority queue only on the facets and not the events. Then, we can define the radius of a facet to be the radius of the minimum event in that facet. Likewise we can define the angle of a facet to be the angle of the minimum event in that facet. Thus, in each iteration of the main loop we only need to remove one item from the priority queue. Also, we can do better than a priority queue. Since all we care about is the minimum event, a heap would be more efficient.

The other issue in implementing the events is how to recompute the events of a facet after adding a new source image to the facet. The starting point is to just recompute the entire Voronoi diagram each time and check each pair of Voronoi cell and ridge for intersection. This can be improved in a number of ways. First, we could compute the Voronoi diagram incrmentally. Better yet, it turns out we don't actually need all of the information in the Voronoi diagram. Instead, we only need to know which source images are closest to some points on the interior of a ridge. Thus, for each existing event we could store the $t$'s corresponding to the endpoints of the intersection of the Voronoi cell and the ridge (here the $t$'s are as in the discussion of the ridges).

Then, when we add the new source image all we need to do is update the $t$'s for each event based on how much the new source image "cuts off". Then, we can check to see if the new source image corresponds to any event.

4.4. **My Implementation.** In my implemenation, I did forego the calculation of the Voronoi diagram when recalculating the events in a facet after adding a source image. However, I did not use any sort of time saving mechanisms for finding the minimum event (I actually just used a Java HashSet for the events).

I also avoided the use of a DCEL. While a DCEL probably would have made the 3-dimensional implementation faster, my longterm goal is an implementation in arbitrary dimensions. And in arbitrary dimensions it's not clear to me how to extend a DCEL.

## 5. Computations with the Source Poset

This section describes two variations of the main algorithm that essentially use the source poset.

5.1. **Voronoi Diagrams of the Boundary of a Convex Polytope.** One use of the source poset is to find the Voronoi diagram of the boundary of a convex polytope. Let $P$ be a convex polytope and let $S$ be a set of points on the boundary of $P$. We can run the main algorithm with one change. Instead of initializing the source images with just the one point $p$, initialize the source images with all of the points in $S$. Now, run the main algorithm as found above. When finished, we need to compute the Voronoi digram of the source images in each facet. The intersection of this Voronoi diagram with the facet is the portion of the Voronoi diagram of the boundary of $P$ on the facet.

5.2. **Unfolding a Convex Polytope.** Another use of the source poset is to unfold a convex polytope. Let $P$ be a convex polytope and $p$ a source point on the boundary of $P$. We need to make one change to the main algorithm. With each source image $s$ maintain an ordered list of the ridges that source image has been folded over. To do this, initialize the list for $p$ as empty. Then, each time a source image $s$ is folded over a ridge $R$, add $R$ to the end of $s$'s list of ridges. Now, run the main algorithm with this change. When done, for each facet compute the Voronoi diagram of the source images in the facet. For each Voronoi cell: intersect with the facet and then unfold this intersection into the affine span of the facet containing the source point $p$ (the unfolding can be done by composing the inverses of the folds along each ridge in the ridge list of each source image). The result will be the unfolding.

The unfolding can be used to find the shortest path from $p$ to any other point $t$ on the boundary of $P$. Simply compute the foldout of $P$ with source point $p$ and locate the image of $t$ in the foldout. Refolding the line segment from $p$ to $t$ back into $P$ gives the shortest path.

## 6. Future Plans

The future goal is an implementation for unfolding arbitrary-dimensional convex polytopes. I'm actually not sure what the applications in dimensions 4 and up are. However, the ability of unfolding a 4-dimensional convex polytope into $\mathbb{R}^3$ could be useful in visualizing 4-dimensional objects.

## 7. Documentation

The code can be found at: www.math.umn.edu/∼byrnes/CSci8442
The relevant files are:

- Event.java
- Facet.java
- Inequality.java
- Point.java
- Ridge.java
- unfold.java

To run the program, first edit unfold.java to contain the correct data.
To compile use the command: "javac unfold.java"
To run, use the comman "java unfold"
The output is the list of source images.

## References

[AA97] Agarwal, Pankaj K.; Aronov, Boris; O'Rourke Joseph; and Schevon, Catherine A. Star unfolding of a polytope with applications, *SIAM Journal of Computing.* **26** (1997), no. 6, 1689-1713.

[MP03] Miller, Ezra and Pak, Igor. Metric combinatorics of convex polyhedra: cut loci and nonoverlapping unfoldings. Available at: http://www.math.umn.edu/∼ezra/

[MM87] Mitchell, Joseph S. B.; Mount, David M.; Papadimitriou, Christos H. The discrete geodesic problem, *SIAM Journal of Computing.* **16** (1987), no 4, 647-668.

[SS86] Sharir, Micha and Schorr, Amir. On shortest paths in polyhedral spaces, *SIAM Journal of computing.* **15** (1986), 193-215.