UNIVERSITY OF WATERLOO

CS 759 Report: Killing Time

Theo Belaire
20415730
Bryan Coutts
20428420

April 7, 2015

This report outlines a solution to the problem of finding a smooth, connected "blob" which contains a specified set of points, and excludes all other points. Objectives are to maximize the distance from interior points to the boundary of the blob, maximize the distance from exterior points to the blob, and to make the blob as smooth and aesthetically pleasing as possible. The source code for our implementation can be found at `https://www.github.com/b2coutts/blob`. A gallery of comb inequalities from a 100-point Euclidean TSP instance, courtesy of Bill Cook, can be found at `http://csclub.uwaterloo.ca/~tbelaire/bico/gal`.

We will use the term *included point* to refer to points that are to be included inside the blob, and *excluded point* to refer to points that must be on the exterior of the blob. The *polygon* will be a (not necessarily convex) shape with we modify throughout the algorithm.

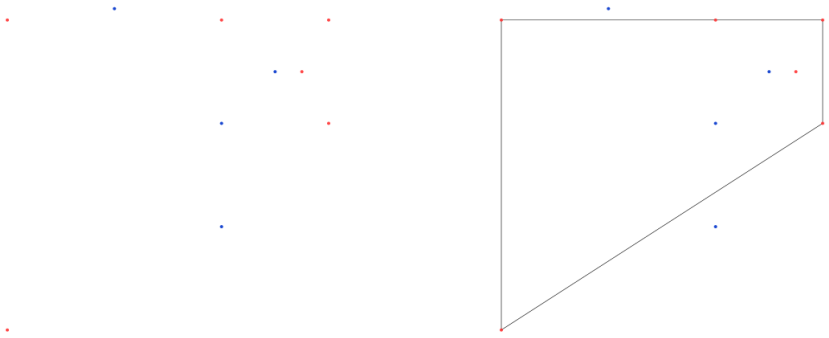
Our algorithm works as follows:

1. Take the convex hull of the included points.
2. Fix the convex hull, so that no excluded points are in its interior.
3. Compute the radius for each point.
4. Induct nearby points to the polygon.
5. Remove "crossover" points.
6. Connect the circles of the vertices of the polygon into a blob.

1. GENERATION OF THE CONVEX HULL

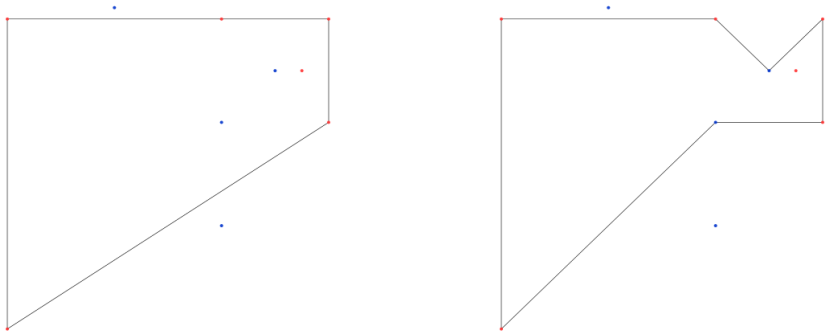
Our code uses the giftwrap algorithm to generate the convex hull. We begin with the leftmost, then lowest, point, and find the first point we hit by casting a ray north and rotating clockwise. Remembering the direction we left off at, we do this again from the new point, and repeat the process until we arrive back at the starting point. The result is shown in Figure 1.1.

At the conclusion of this step, we have a list of included points in clockwise order, which forms our polygon P. Currently it is convex.



(a) Before (b) After

Figure 1.1: Computing the convex hull



(a) Before (b) After

Figure 2.1: Removing excluded points

2. Point Exclusion

In this step, we will adjust P, so that no excluded point is in its interior. We will do so by removing triangles from P; this will cause it to no longer be convex.

To test if an excluded point is within P, we use the Jordan Curve Theorem. One of its consequences is that if we cast a ray from the point in any direction, it will cross an edge of P an even number of times if and only if the point is outside P. This is computationally simple to check.

For each excluded point p in the interior of P, we find the edge closest to p. We consider only the edges whose endpoints p is between. We then insert p into our list of polygon vertices, between the endpoints of this edge. This removes the triangle formed by the edge and p from P.

3. Calculating Radii

The *radius* of a point determines how large a circle is used when drawing the blob around it. If all the radii were very small, the blob would appear the same as the polygon. However, the radii cannot be too large, otherwise the circles draw around included and excluded points will intersect with each other. Hence, the radius of an included point is bounded by its distance to an excluded point, and vice-versa. Moreover, there are situations in which the radius of an included point must be bounded

by its proximity to another included point, as illustrated in Figure 3.2.

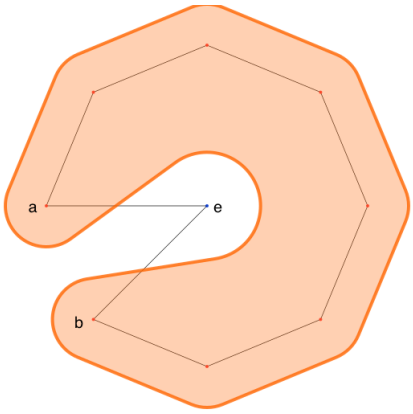


Figure 3.1: If the radii of a, b are too large, the blob will self-overlap

For included points a, b, a must bound the radius of b if and only if the line segment from a to b is contained entirely within P. Similarly, for excluded points a, b, a must bound the radius of b if and only if the line segment from a to b is disjoint from P. We check this condition by checking if any edges of P intersect with the line segment from a to b.

Given this notion of which points bound each others' radii, we choose the radius of a point to be $1/3$ of its distance to the nearest bounding point. This ratio must be at most $1/2$ to avoid overlap. The smaller the ratio is, the thinner and blockier our blob will be. $1/3$ allows for a curvy blob, while avoiding some pinching effects observed with higher ratios.

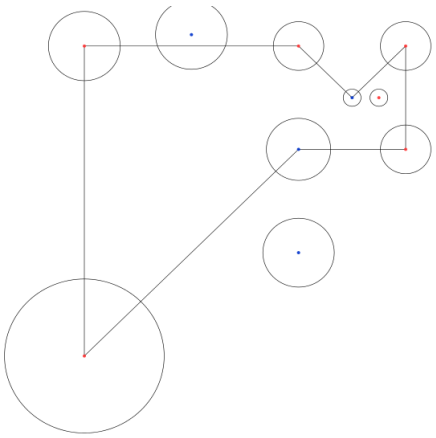


Figure 3.2: The radii of each point

4. Inducting Nearby Points

Once we have all the radii, we almost know the final shape of the blob. However, there may be excluded points near the edge of the polygon, whose circles (or even the points themselves) may intersect the blob, as seen in Figure 4.1. There may also be included points near the border, whose circles would intersect the exterior of the blob. We add all such points to the polygon.

We iterate over all points p that are not vertices of P. Then, for each edge $e = (a, b)$ of P, we check if the line from the edge of the circle of a to the edge of the circle of b intersects the circle of p. If it does, we add p to P, between a and b (effectively adding the triangle with vertices a, b, p to P).

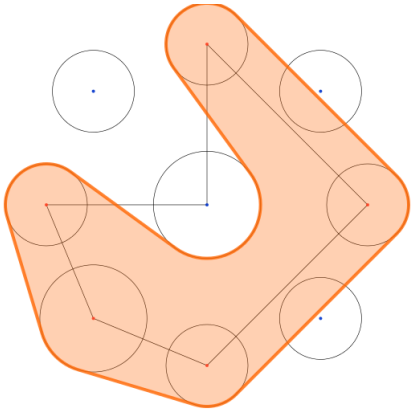
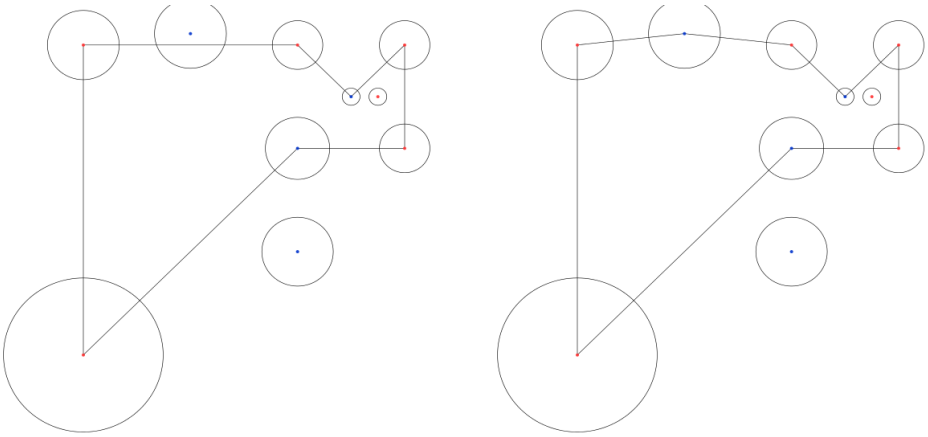


Figure 4.1: The circles of the bottom-right, top-left points are cut by the blob



(a) Before (b) After

Figure 4.2: Removing excluded points

5. Uncrossing

In adding these points, we may have created some undesirable loops, as seen in Figure 5.1. We must do an extra pass over the vertices of P to identify these cases and eliminate them. Such cases involve visiting a vertex that does not need to be visited; by simply finding and removing the vertices around which the blob crosses itself, we can eliminate this issue.

6. Blob Drawing

Finally, we draw the blob itself. We do this by connecting the circles of the vertices by tangent lines. For each edge $e = (a, b)$ of P, we apply some elementary linear algebra to find the line tangent to the circles of (a, b). If a, b are both included, the line is on the outer edges of their circles. If a, b are both excluded, the line is on the inner edges of the circles. Otherwise, the line weaves between a and b. This forces included points to be in the interior of P, and excluded points to be disjoint from P.

Conclusions and Further Study

The current implementation is generally capable of producing nice-looking blobs, and is suitable for use in visualizing sets and combs. However, there are some edge cases that it does not handle optimally. Moreover, the generated blobs could be made smoother and more aesthetically pleasing by choosing radii more flexibly, and by introducing "dummy" points to the polygon. Moreover, further effort could be put into optimizing the runtime of the code. The strategy for constructing the blob

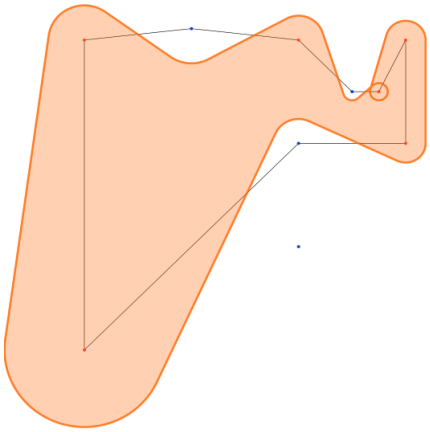
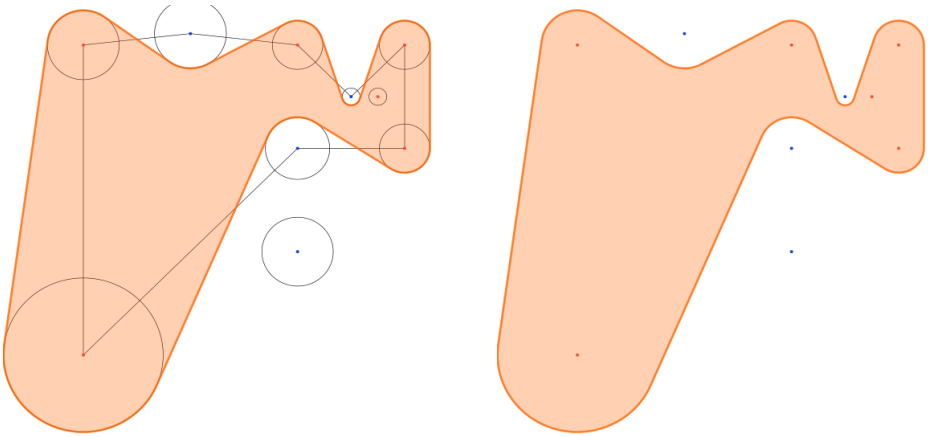


Figure 5.1: A section in which the blob crosses itself



(a) Rough (b) Clean

Figure 6.1: Drawing the final blob

could be changed based on the density and layout of the input data. Multiple strategies could be employed, and an algorithm used at the end to take the best result, via metrics such as volume, girth, and point-to-boundary distances.