Concave Hull Kernel

Joel Stansbury
CS7375

Abstract

This paper introduces a method for calculating the non-convex (or concave) hull of a 2-dimensional cluster of points. The first of two interesting qualities is that the core computation can be expressed as a convolutional kernel, potentially allowing for GPU acceleration. The second is that the expensive portion of the algorithm is only executed on each point of the polygon, all other points in the cluster are ignored. The algorithm builds off an imaginary scenario in which a rod is pivoting from one end around a point on the hull. When it collides with another point, the rod will slide backwards until the other end is in contact with the new pivot and begin rotating around the new pivot. This process repeats until the rod arrives back to the starting location.

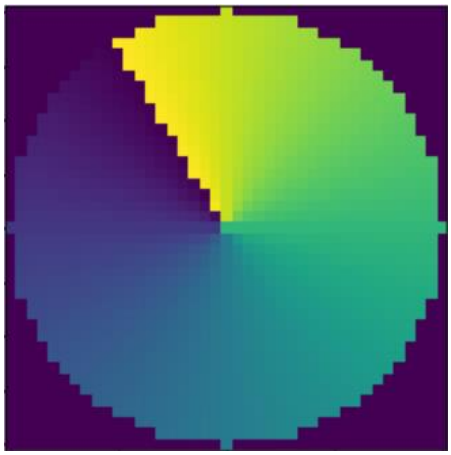


figure 1: Concave Hull Kernel

Introduction

Bounding hull problems can be summarized as finding the polygon which completely surrounds a set of points while simultaneously minimizing some property of the polygon itself (surface area or maximum edge distance for example). Typical applications usually involve some form of computational image processing or handling 3D point cloud data. One such application is image segmentation, wherein pixels are clustered based off color, optical flow, location, and/or other qualities, and a mask is created for each cluster. In some cases, a pixel may be incorrectly excluded from a group leading to a hole in the mask. If the group is sufficiently dense, this noise can be mitigated by a simple blur + threshold, at the expense of detail in the boundary edge. If edge detail must be preserved, or sparsity of the cluster demands an unacceptable amount of blurring, an alternative approach to obtaining a continuous mask is to compute the bounding hull of the cluster directly from the coordinates of the members.

The convex hull is "the minimum convex polygon for which all points are either inside this polygon or at its border" [1].

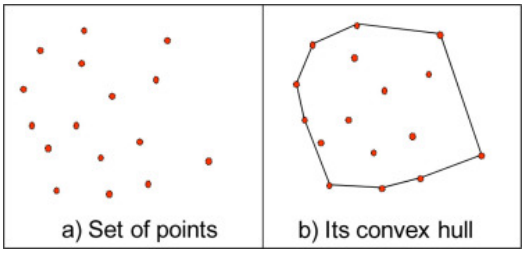


Figure 2: Convex hull example (figure 5.7 from Geographic Knowledge Infrastructure) [1]

Notice that none of the internal angles exceed 180 degrees, this is what makes the hull *convex*. A concave hull, on the other hand, does allow angles above 180 degrees.

Figure 3: Concave Hull Example.

Approach

In the abstract of this paper, we laid out an imaginary scenario of a rod rotating around the hull. Here we'll address some of the implementation details required for this algorithm to function.

Critical Assessment of the General Strategy

How do we get to the edge?

During the experiments conducted for this paper, the edge was found by sorting the points. It is not infeasible, however, to conceive of a kernel which enables the algorithm to find the left-most point. The following figure shows a kernel defined by setting the value of each point (x,y) where the point (0,0) is at the center, equal to (-x/(1+abs(y)). Where, the size of this matrix is defined by the chosen rod length. The application of this kernel would involve

1. starting from a random point in the cluster,
2. cropping out a suitably sized matrix around that point,
3. multiplying (bitwise) the cropped portion with the go_left kernel shown below,
4. finding the argmax of the result,
5. and navigating to that new point until the max of the bitwise multiplication is <0.

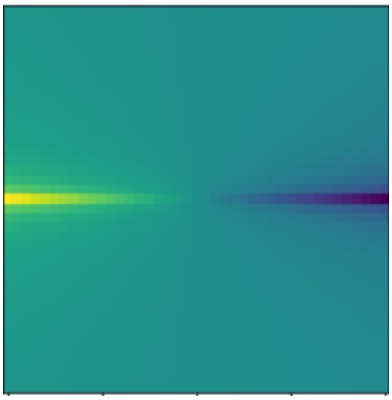


Figure 4: Go left kernel

At this stage, we could either be on the right side of a hole, or the left edge of the cluster. Unfortunately, there is not currently a good way to ensure that you are in fact on the outer edge so it is necessary to validate the polygon afterwards. This can be accomplished by accumulating the change in the leading angle to determine if the polygon was constructed in a clockwise direction. If the angles sum to -2π, then we know that the polygon was constructed in the anticlockwise direction and must reinitialize the algorithm from another random point.

How do we know where to start the rotation?

This one actually has a simple answer. If the rod is always rotating clockwise, and the previous point was at angle Θ counter-clockwise from the positive x-axis, then the rod will start rotating at some angle which is less than Θ. See figures 5 and 6 for a visual explanation.

What happens if the rod is too short?

Currently, this is sometimes catastrophic. We have not categorized all the ways in which this scenario will cause problems for the algorithm, but suffice it to say, it will occasionally discover

loops (both finite and infinite) which lead to a poor hull boundary. Resolving this is a topic of future research. The current solution is to simply increase the rod length. Seeing as how the rod length determines the size of the kernel, this is fairly expensive. The algorithm complexity increases as $(2 * m + 1)^2$, where m is the rod length.

How is a Cycle Detected?

This is a bit tricky as well. The most obvious marker is landing on a pivot that has already been seen. This, however, is insufficient. Take a figure 8 configuration of points for example. The center point connecting the two lobes will be seen twice before a full cycle has been explored. To account for this, we continue the search until 10 re-hit events occur. This seems to be ok but is a potential subject for future work. A better solution may be to require multiple re-hits in a row.

Implementation Details

Defining the Sweep Kernel

This is defined in a similar manner to our go_left kernel in figure 4. The goal is to find the first point that a rotating rod would contact when rotating around the current pivot. To achieve this we utilize trigonometry to set each pixel of the kernel equal to the angle dΘ that the rod rotates before overlapping that pixel.

Rotation of the Sweep Kernel

Rotating this kernel is surprisingly straightforward. Since every pixel value is just the angle of rotation required to reach it (if a rod started at some angle Θ and rotates clockwise). To change the initial position of the rod (Θ to Θ + offset) we simply add the offset to all values of the kernel. Any values that exceed 2π are decreased by 2π, and any values below 0 are increased by 2π.

Application of the Sweep Kernel

Here is an example of the boolean matrix around the current pivot.

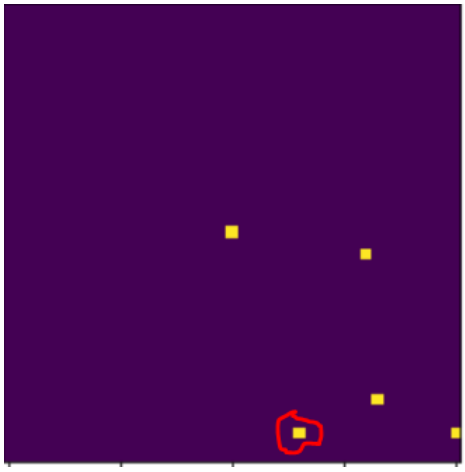


Figure 5: Vision matrix. The current pivot is in the center of the matrix, while the previous pivot is circled in red.

Here is the sweep kernel at this point.

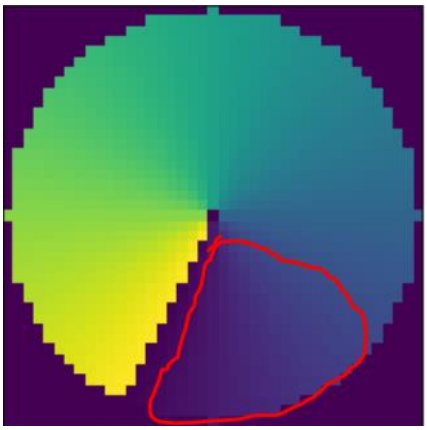


Figure 6: Sweep kernel at current location.

The area circled in red (figure 6) indicates the approximate location of the previous pivot. At this stage, we don't care where exactly the previous pivot was, only that it is behind the leading edge of the sweep kernel. In this implementation we always set the sweep kernel to start 45 degrees past the last pivot. This angle was chosen because any point within the first 45 degrees would have been detected by the previous sweep. Notice also that the center of the kernel is set to zero, this prevents the pivot from being considered by the sweep.

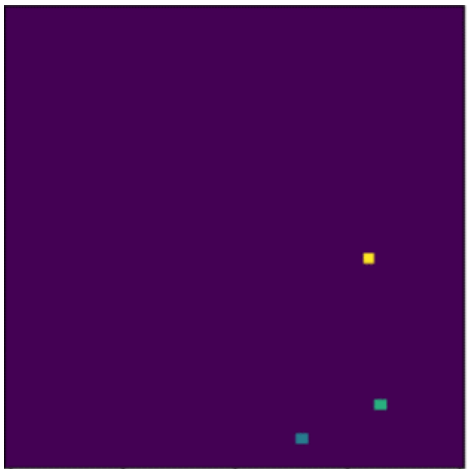
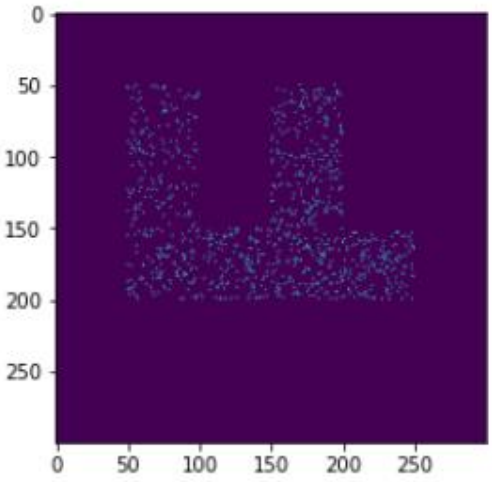


Figure 7: post bitwise multiplication

After performing the bitwise multiplication, we see that the previous pivot is dim because this portion of the sweep kernel is closer to zero. There are also two points which are invisible, one is the current pivot (in the center), and the second is a point just outside the maximum range of the sweep near the lower righthand corner (see figure 5).

Results

We compared this algorithm against an implementation of alphashape on two examples of the “F” configuration of random points. The first example contained 883 points in total. Both algorithms returned acceptable boundaries.



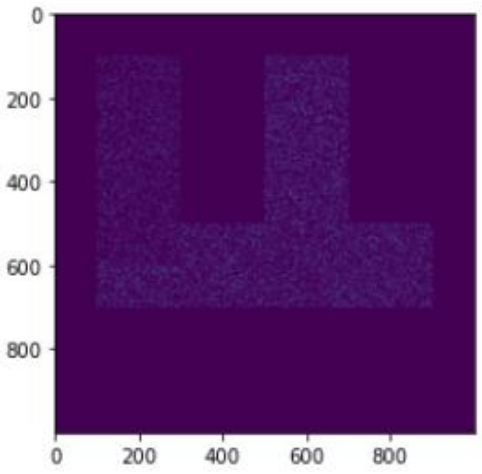

```
t0 = time()
poly = Polygon(planky(points, 11))
print(f"planky done in {time()-t0}")
poly
```

```
planky done in 0.00899648666381836
```

```
t0 = time()
a = alphashape.alphashape(points, 0.1)
print(f"alphashape done in {time()-t0}")
a
```

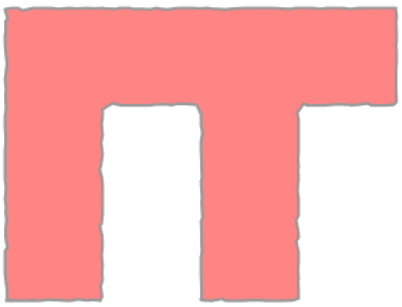
```
alphashape done in 0.18599629402160645
```

The second example is the same shape and density, but is a larger image, so has more points to consider (13936 points in total).




```
t0 = time()
poly = Polygon(planky(points, 24))
print(f"planky done in {time()-t0}")
poly
```

```
planky done in 0.05574655532836914
```



```
t0 = time()
a = alphashape.alphashape(points, 0.1)
print(f"alphashape done in {time()-t0}")
a
```

```
alphashape done in 3.099370002746582
```

Conclusion

At 900 points, our algorithm took roughly 4% of the time that alphashape took. At 14,000 points this ratio continued to decrease such that our algorithm took just 1.9% of the time that alphashape took. Further testing is undoubtedly required, and there are some significant problems which still need to be addressed, but it does appear that the lower-order dependence on cluster size may provide an advantage under the correct circumstances.

References

[1] Robert Laurini. Section 5.1.5 from Geographic Knowledge Infrastructure. *Applications to Territorial Intelligence and Smart Cities*. 2017

[2] Ken Bellock. Implementation of the Alphashape Algorithm on *https://github.com/bellockk/alphashape*.