Fast Polygon Visibility Graph Implementation

Spencer H. Eanes
St. Olaf College
Math 282
eanes1@stolaf.edu

May 20, 2019

Abstract

This paper will describe an implementation of the $O(n^2 \log n)$ polygon visibility algorithm described in [1] and [2]. It is implemented in R, particularly with use of the Simple Features (SF) package, and visualized through an RShiny web application [1].

1 Introduction

Consider a set of polygons in a plane. The problem of determining which vertices have an unobstructed path to other vertices is of particular interest to fields such as robotics.The solution to such a problem could be used, for instance, to compute the optimal shortest path of a robot through a field of obstacles. As with most algorithms, it is desirable to compute this is quickly and efficiently as possible.

1.1 The Naive Approach

Consider a set of m polygons, $P = \{P_1, P_2, ..., P_m\}$ in a plane, with vertices $W = \{w_1, ...w_n\}$ and edges $E = \{e_1, ..., e_n\}$. The simplest way to solve this problem would be to consider each vertex, w_i, against every other vertex, w_j, and check each of these combinations against each polygon edge. That is, check if the line segment $w_i w_j$ intersects any edge $e_k \in E$. If there is no intersection, then $w_i w_j$ is a visibility edge of this set. However, this algorithm takes $O(n^3)$ time, as n vertices are checked against n other vertices which are checked against n edges. Can it be done faster by capturing information as it runs?

1.2 A Faster Approach

[1] proposes an implementation that runs in $O(n^2 \log n)$ time. The idea of this algorithm is that for each vertex w_i, all other vertices w_j are sorted according to angle from horizontal line from w_i to the line segment $w_i w_j$. A binary search tree is then used to store the edges of polygons that could possibly intersect new visibility edges. Since lookup, insertion and deletion in this tree takes $O(\log n)$, this reduces the run time to $O(n^2 \log n)$. Further details can be found in [1] pages $326 - 330$ and [2].

2 Using R

I chose to implement the algorithm in R because I wanted to create an RShiny web app for visualization. There proved to be both advantages and disadvantages to this. I am unfortunately unfamiliar with creating classes in R, so I had to approach this problem functionally and through the use of R packages. Packages were able to provide out of the box functionality that I needed, but this also meant learning how to use them.

[1] Application available at math282.spencereanes.org

2.1 Simple Features Package

R has a CRAN supported package, SF, [3], simple features, which is "a standardized way to encode spatial vector data." Unfortunately, it has a rather steep learning curve and has relatively sparse documentation. Many of the tutorials I found were aimed at geo-spatial data, such as large scale maps. I chose to use this package as it has many useful features, in particular intersection detection. There is a single set functions that can detect intersection, overlapping, touching or crossing for points, line segments, and polygons, as well as lists of these, which have their own class. Though this meant learning to use the package, it also meant I didn't have to create a class or function to do this for me. SF also native support in ggplot2 through the geom_sf function, which was very useful for the visualization.

2.2 R Binary Search Tree Package

As far as I could find, R does not have any sort of native or CRAN supported package for binary search trees. There is the data.tree package, but it doesn't support binary tree construction and lookup in the ways I hoped. Instead, I found a package, rbst, which is only available on Github (not through CRAN), [4]. The package supports side-effect free binary search tree with guaranteed logarithmic time for insert, delete and retrieve. It "achieves perfect balance (and the resulting speed guarantees) by implementing a left-leaning red-black tree." I used this package for all my binary tree needs through this project.

3 Implementation Details

While the pseudo-code for the this algorithm available in [1] does not look particularly difficult to implement, the devil is in the details. Please note that in this section I will denote vertices by a number, and edges by two vertex numbers separated by a dash.

3.1 Creating Polygons

The first issue that had to be addressed was how polygon data would be stored and transferred between functions. In the RShiny application, the polygons are represented simply as a dataframe of points with three columns, x and y coordinates, and an identifier for the polygon they belong to. The assumption in this case is that edges exist between adjacent vertices in the dataframe, and the first and last. This displays perfectly with the ggplot geom_polygon function, but does not play nicely with other functions. To create a more suitable universal object that holds all the information needed, when the "complete current polygon" button is pressed, all the vertices of the current polygon are used to create a SF Polygon object, and is stored in a list. When the visibility graph is computed, this list is transformed into a MultiPolygon object.

3.2 Building the Tree

Handling the tree was probably the trickiest part of this whole exercise. The pseudo-code in [1] simply says that when finding visibility edges from a vertex v, polygon edges that are clockwise of the half line from v to each other vertex w_i should be added to the binary tree, and edges counterclockwise from vw_i should be removed. However, there are many nuances to performing this properly. The first was determining which edges leave any given vertex, as this is not stored within the SF objects. To solve this problem, I created a function that takes a dataframe of x and y coordinates and appends four additional columns two x and and two y coordinate columns for each of the two other vertices it's connected to.

The next issue was how to give edges a unique key in the tree that still allowed them to be sorted. Since we only ever retrieve the minimal key (the leftmost leaf), I initially decided to simply key edges with an integer as they appeared. However this presented the issue of looking them up later. To tackle this, I created a hash table, of which the key was the edge vertices as a string, and the stored hash table value was the lookup key for the binary tree. Though a "hacky" work-around, it works because hash table insert and lookup is O(1). Additionally, the vertices as a string key is unique since two edges should never have the same start and end point. Each edge had to be added twice to the hash table, once for each ordering of the vertices. However, this created incorrect visibility edges.

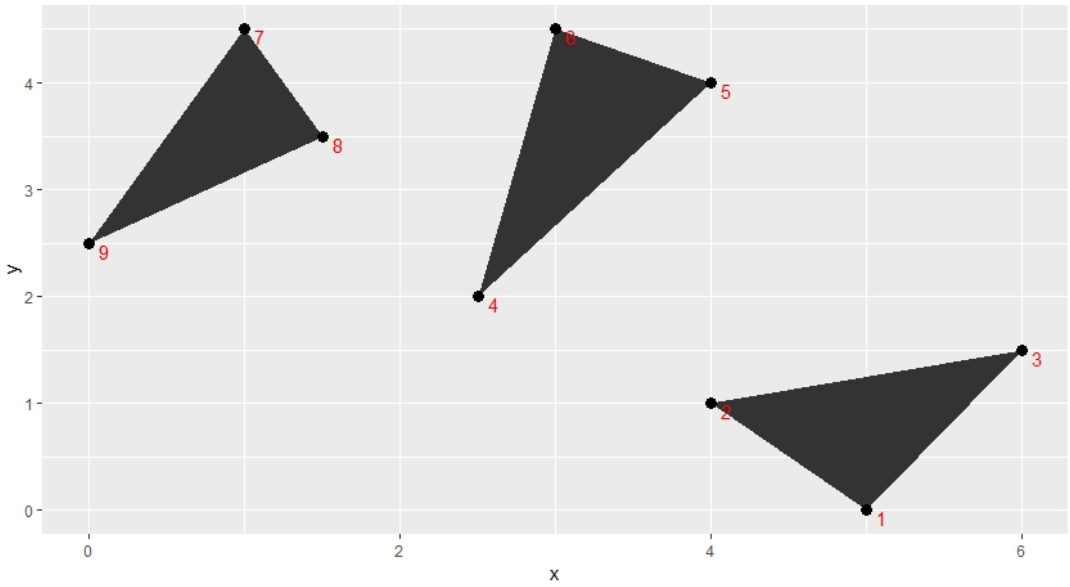

Figure 1: A set of polygons for which the VisibleVertices algorithm will fail if tree edges are sorted strictly by order edges are inserted.

3.2.1 An Error Due to Wording

Using the implementation described in 3.2, I discovered an error in testing. An example that would fail can be seen in Figure 1. In this example, the code would produce an incorrect visibility edges from vertex 2 to vertex 8. The pseudo-code says to insert polygon edges into the binary tree "in the order which they are intersected" (note that this really means crosses, not intersects, as a visibility edge could intersect perfectly at a vertex and still see something beyond it, discussed further in 3.3.1). Let's follow the VisibleVertices algorithm as described in [1] page 327 for figure 1 to flesh out this issue. Let T be the binary tree, and V be the set of visible vertices from our sample vertex, 2. First, vertices 2 and then 1 are added to the visible vertices list, while no polygon edges are added to T. Then 9 is added as it intersects no edges of the tree, and edges $9-8$ and $9-7$ are added to the tree in that order. Vertex 4 is checked next, found visible, and $4-5$ and $4-6$ are added to the tree with higher indices than $9-8$ and $9-7$. Vertex 8 is considered next, and checked against the leftmost leaf of T, which is currently $9-8$. In this case VisibleVertices returns that vertex 8 is visible since $2-8$ doesn't cross $9-8$. In order to correctly detect that it is not visible, vertex 8 would need to be checked against edge $4-5$ or edge $4-6$. So how do these get sorted into T in a position lower than the previously inserted $9-8$?

3.2.2 Tree Sorting by Edge Distance

My solution to this issue was to key and sort edges in the tree T by the distance from the vertex in question to the midpoint of the polygon edge being inserted. I think an argument could be made that pseudo-code "in the order by which they are intersected" could be interpreted to mean closer polygon edges should be inserted with lower key values. However, this seems like a loose interpretation of the wording of the pseudo-code. On the other hand it resolved all the issues I was experiencing.[2] This does introduce one scenario for which the algorithm may fail - the case where the midpoints of two edges are equidistant from some other vertex of the polygons. However, if we assume this will not occur, which is a farily safe assumption for use in the Shiny app, then there are no failing situations.

[2]I would like to note that I don't have a proof that this will always work, and there is a possibility that I am misunderstanding some detail of this algorithm that would resolve this issue. However, as best as I can understand, ordering the tree by edge distance is the only way to resolve the issue presented in 3.2.1.

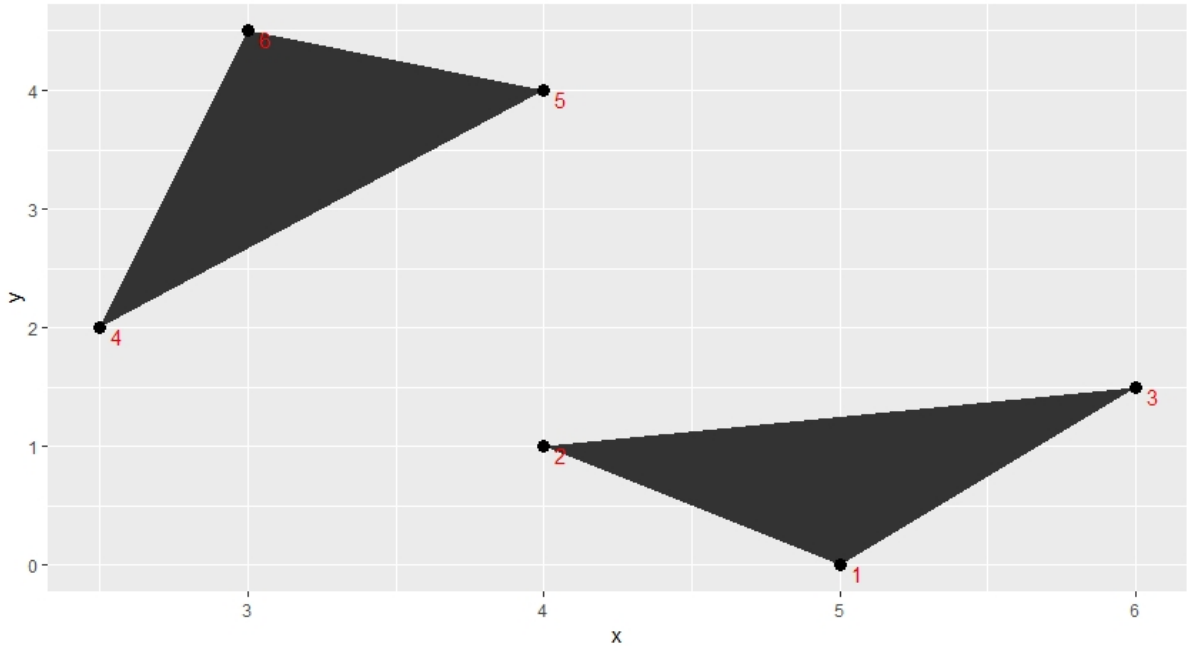

Figure 2: Visibility edge $2-5$ will intersect the leftmost polygon edge $4-5$ despite being visible.

3.3 Special Case

In the conditions of the visible algorithm detailed in [1] and [2], there is a special case which is not explained but deserves mention for both future implementations and the difficulty it caused me in debugging.

3.3.1 Intersection vs. Crossing

The Visible algorithm described [1] page 329, lines 5-6 says that when determining visibility from a point p to an vertex w_i, if the segment pw_i intersects any part of the edge in the leftmost leaf of a binary search tree T, then the leaf is not visible. However, it is possible for pw_i to intersect only at one of the end points and still have w_i visible. Consider Figure 2, and the visibility edges from vertex 2. When we examine the second to last vertex ordered by angle, vertex 5, the leftmost edge of T will be edge $4-5$. Edge $2-5$ does intersect edge $4-5$ at point 5, making it not visible according to the cut-and-dry Visible algorithm, despite clearly being visible to a human. To modify this, I made this if statement check for crossing rather than intersection.

4 Conclusion

Implementing the details of this algorithm from pseudo-code was very informative. I had to make many considerations about storing spatial data, communicating data between inter-reliant functions, and most of all debugging and discovering details not covered in pseudo-code. This was also my first experience using RShiny (thus the roughness of the app), but I liked it a lot and have hopes to use it more in the future.

References

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[2] Philip Kwok. An $o(n^2 \log n)$ algorithm for computing visibility graphs.

[3] Edzer Pebesma. Simple Features for R: Standardized Support for Spatial Vector Data. *The R Journal*, 10(1):439–446, 2018.

[4] tarakc02. Persistent balanced binary search tree in r, 2016.