Project Report

Rikil Gajarla (2017A7PS0202H)

June 2020

Convex Hull

Introduction

Given a set of points P, the Convex Hull of P, denoted conv(P) is the smallest closed simple polygon which contains all the points in P. It can be visualized as a tightly snapped rubber band over the given set of points.

We would like to compute the convex hull of given set of points by using a divide and conquer technique. Later, we will also show that the computational complexity of our algorithm is also O(n log n). Note that we cannot do better than is for computing convex hull

How to Run

The src folder contains the source code for the convex hull program. `g++` from the GNU compiler suite is required to compile the program to a executable.

Steps to Compile:

1) `cd` into the src directory
2) Run `g++ main.cpp` which generates an executable called `a.out` in the same directory
3) Run the executable using `./a.out` (on linux)
    1) The executable takes a dataset from command line argument. For example, to use an existing dataset, run `./a.out ../datasets/edge.txt`
    2) If no command-line argument is given, it takes input from the shell directly (stdin)

Input

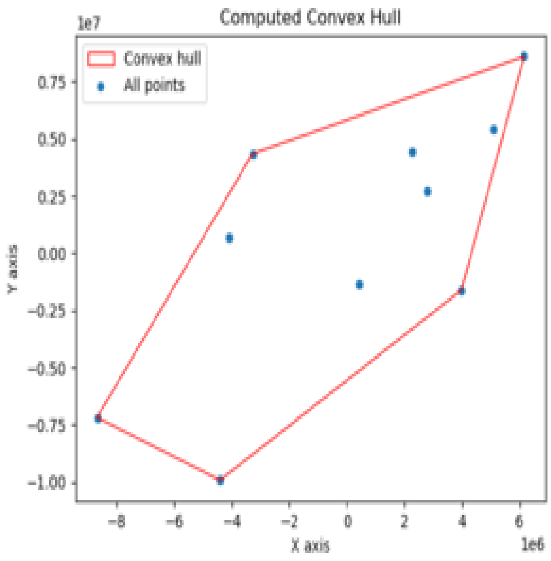
The required file format for the algorithm to work correctly is:

- First line must contain the no of Points to be taken as input by the program.
- Each of next line must contain 2 integers, space seperated denoting the (x, y) coordinates of each point.
- Each coordinate must be of integer type in the range -10^6 to 10^6.
- No of coordinates must be less than 1 Billion.
- **Note**: while using floating point datasets, the output coordinates might slightly differ as given coordinates are stored as floats.

Output

Each line of output contains the coordinates of points present on the convex hull in clockwise order.
The last two lines of output show the time taken to take input in microseconds and the time taken by the algorithm to compute convex hull (also in microseconds).

Example output:



Documentation and Report

Documentation of this algorithm, functions and classes can be found in the `docs` folder in the current directory. Open the index.html file from the docs directory with your preferred browser to go through the documentation

Performance Analysis

Analysis is performed with a system running:

- OS: Arch Linux (64Bit) running Linux Kernel version 5.7.2
- Processor: Intel Core i7 7700HQ
- RAM: 8GB
- Compiler: GNU G++ (GCC) 10.1.0

The following observations are recorded:

Filename	Input Dimentions	Output Diemntions	File read time	Algorithm runtime
edge.txt	5	4	451 microsec	8 microsec
small.txt	7	7	495 microsec	16 microsec
test.txt	34	7	328 microsec	56 microsec
radial.txt	3100	9	1348 microsec	6.15 millisec

Randomly generated points (using python):

Filename	Input Dimentions	Output Diemntions	File read time	Algorithm runtime
10.txt	10	5	314 microsec	16 microsec
20.txt	20	7	495 microsec	31 microsec
50.txt	50	9	427 microsec	108 microsec
100.txt	100	13	348 microsec	165 microsec
500.txt	500	13	572 microsec	935 microsec
1000.txt	1000 (1K)	18	510 microsec	2.1 millisec
3000.txt	3000 (3K)	24	2.49 millisec	5.5 millisec
5000.txt	5000 (5K)	22	2.12 millisec	8.0 millisec
10000.txt	10000 (10K)	23	3.26 millisec	17.6 millisec
50000.txt	50000 (50K)	28	17.4 millisec	80.5 millisec
100000.txt	100000 (1L)	31	39.3 millisec	164.1 millisec

Filename	Input Dimentions	Output Diemntions	File read time	Algorithm runtime
250000.txt	250000 (2.5L)	31	78.1 millisec	448.3 millisec

Using real world datasets:

Filename	Input Points	Output Diemntions	File read time	Algorithm runtime
subway-entrance-ny.txt	1929	16	2.85 millisec	3.81 millisec
parking_meter.txt	15191	16	14.15 millisec	27.04 millisec

Sources of datasets:

- Parking meter dataset
- New York Subway Entrance dataset

It is highly difficult to calculate the time taken exactly up to the microsecond, hence the value might vary on different executions. Also, please note that the time calculated here may vary based on the system load and other background applications.

Algorithm Approach

For more detailed information regarding the algorithm, refer to David Mount Lecture notes from the course CMSC 754 - Computational Geometry

Given set of points P, we would like to first order them according to the increasing x coordinate. For the sake of simplicity, let us assume that no two points will have same x or y coordinate and no 3 points are co-linear.

First, we will recursively compute the upper hull and then similarly compute the lower hull of the given set of points. Finally, we will join the upper hull with the lower hull to complete the convex hull and return/print the clockwise ordering of the points present on the convex hull

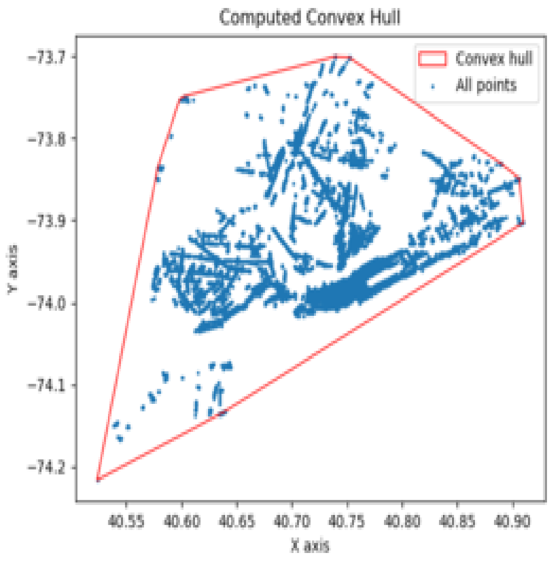
Upper Hull Computation

We divide the input set of points into 2 equal halves and recursively compute the upper hull for both the half. Then, we compute the upper tangent to both the hulls by traversing from the right on left hull and traversing from left on the right hull. We perform orientation tests to determine when the orientation changes on each hull respectively and move upwards a point. We perform this operations until we reach the upper part of both hulls.

All the traversed points on both hulls except the one which lead to tangent are deleted (because all of them lie under tangent, hence lie inside the polygon) and the resulting points of both hulls are catenated to get the upper hull.

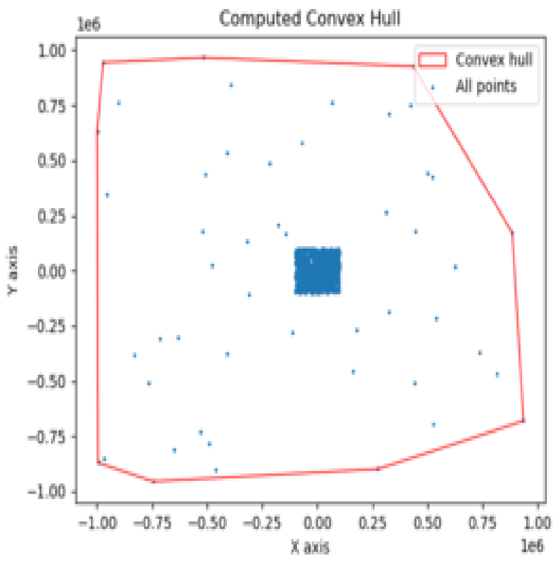
Results

The following is the computed convex hull of the parking_meter.txt dataset:



The red boundary in the above image represents the convex hull given as the output by the program. The blue points are the inputs given to the program.

By analysis of the above divide and conquer algorithm, It can be concluded that the algorithm runs in O(n log n) time complexity. From the above results of testing of various datasets, we can see that the time increase is proportional to the no of points.



The dataset radial.txt is a hand-crafted dataset (3K points) which has high concentration of points (~2.1K) in the center, whereas other randomly generated datasets (for example: 3000.txt)have a uniform distribution of points (the python random number generator uses an underlying uniform distribution). From this we can assume that independant of positioning of points, our algorithm takes approximately the same time to compute the convex hull.

Conclusion

For the last dataset which contains 2.5L points, its worth noting that the algorithm runs in just under 500ms which might appear to be fast. But for applications like graphics, rendering and animations, this poses a huge overhead in terms of repeated computation of convex hull for various objects.

One observation we can make here is that the number of points which are present on the convex hull is usually relatively lower most of the times. So, using an output sensitive algorithms (Ex: Jarvis's march) might help us if our application has fewer vertices on the convex hull.

It's worth noting that convex hull computation cannot be done under O(n log n) which has been proved. Therefore, we can only hope to reduce the constants of complexity. In our algorithm, we repeatedly recurse, hence make many function calls which is computationally expensive. Hence by using algorithms like Chan's or Grahm's, we can do better

DCEL - Doubly Connected Edge List

Introduction

Doubly connected edge list (DCEL), which is also known as halfedge data structure, is used to efficiently represent planar graphs. Using a DCEL structure, we can easily manipulate the topological information of the planar graph such as vertices, faces and edges.

In general, a DCEL conatins a book-keeping record of all edges, vertices and faces of the graph. Each record individually can contain information of all it's incident geometric objects. We will look at how a DCEL stores this information in Data Structure Approach section.

How to Run

The src folder contains the source code for the DCEL data structure. `g++` from the GNU compiler suite is required to compile the program to a executable.

Steps to Compile:

1) `cd` into the src directory
2) Run `g++ main.cpp` which generates an executable called `a.out` in the same directory
3) Run the executable using `./a.out` (on linux)
    1) The executable takes a dataset from command line argument. For example, to use an existing dataset, run `./a.out ../datasets/1sq.txt`
    2) If no command-line argument is given, it takes input from the shell directly (stdin)

Input

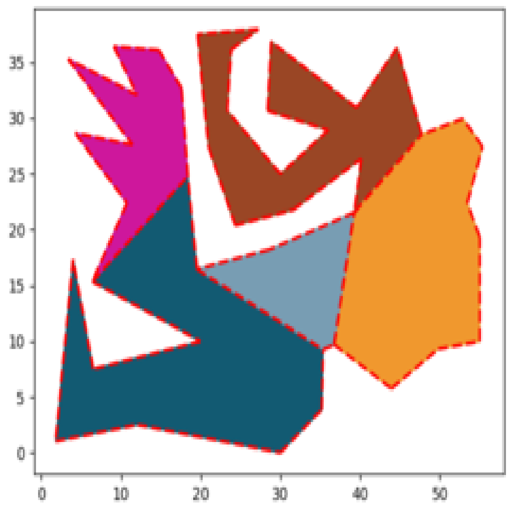
The required input format for the algorithm to work correctly is:

- Input can be given both from file (via command-line args) or stdin
- First line must contain the no of Edges to be taken as input by the program.
- Each of next line must contain 4 integers, space seperated denoting the (x1, y1), (x2, y2) coordinates of endpoints of each edge.
- Each coordinate must be of integer type in the range -10^8 to 10^8.
- Number of coordinates must be less than 1 Billion.

Output

Output of the algorithm prints all the HalfEdges incident to each face by traversing its representative half edges. All the faces outputed by the program can also be verified by running it through the plotting diagram.
The last two lines of output show the time taken to take input in microseconds and the time taken by the algorithm to compute DCEL (also in microseconds).

An example of output of dataset random4.txt:



Documentation and Report

Documentation of this algorithm, functions and classes can be found in the `docs` folder in the current directory. Open the index.html file from the docs directory with your preferred browser to go through the documentation

Performance Analysis

This analysis does not include the time taken to show output on terminal. Analysis is performed with a system running:

- OS: Arch Linux (64Bit) running Linux Kernel version 5.7.2
- Processor: Intel Core i7 7700HQ
- RAM: 8GB
- Compiler: GNU G++ (GCC) 10.1.0

The following observations are recorded:

Manual generated test files:

Filename	Input Vertices	Input Edges	Input Faces	File read time	DCEL Creation Time
1sq.txt	4	4	2	505 microsec	27 microsec
4sq.txt	9	12	5	396 microsec	46 microsec
random1.txt	10	12	4	257 microsec	52 microsec
random3.txt	23	26	5	175 microsec	84 microsec
random2.txt	23	37	16	192 microsec	102 microsec
a.txt	31	33	4	150 microsec	109 microsec
random4.txt	44	48	6	364 microsec	165 microsec
house.txt	71	81	12	250 microsec	300 microsec

Using real world datasets:

- Road network of hyderabad, source: GPS coordinates filtered from osm(openstreetmap) data on hyderabad
  - hyd_1.txt: contains road network having tag of *highway* set as ‘trunk’
  - hyd_2.txt: contains road network having tag of *highway* set as ‘trunk’ and ‘primary’
  - hyd_2.txt: contains road network having tag of *highway* set as ‘trunk’, ‘primary’ and ‘secondary’

– hyd_2.txt: contains road network having tag of *highway* set as 'trunk', 'primary', 'secondary', and 'motorway'

Filename	Input Vertices	Input Edges	Input Faces	File read time	Algorithm runtime
hyd_1.txt	685	733	58	2.4 millisec	1.68 millisec
hyd_2.txt	1787	1912	229	3.5 millisec	107.5 millisec
hyd_3.txt	2959	3198	519	5.4 millisec	276.9 millisec
hyd_4.txt	3612	3850	532	24.3 millisec	337.8 millisec

Data Structure Approach

The following geometric objects are used in DCEL:

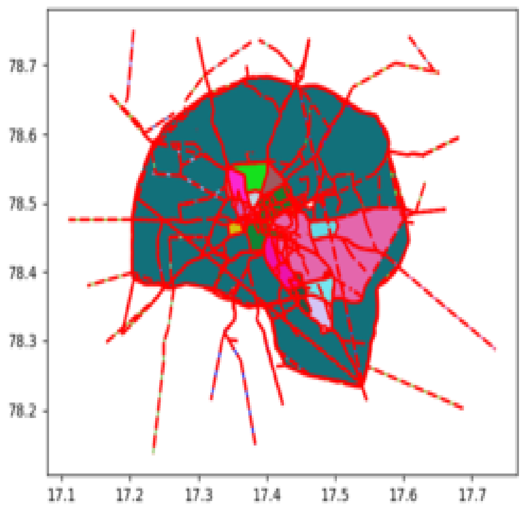
- **Vertex**: Object to store vertex coordinates and any one of its incident halfedges as its representative.
- **Edge**: Object to store its vertex endpoints.
- **Face**: Object to store a unique face-id and one of its incident halfedges as its representative
- **HalfEdge**: Object to store a directed halfedge which has pointers to all its incident geometric objects (Vertex, Face, next and previous halfedges and its twin halfedge)

This implementation uses a Hash Map to store list of all incident halfedges for each vertex. This Hash Map uses vertex pointer as key and stores a list of halfedges as value. Our DCEL also has a list of Vertices and Faces which can be used to iterate through all Vertices and Faces quickly.

A half-edge has a pointer to the next half-edge and previous half-edge of the same face. Each half-edge has a single face as its representative Face (which is present towards left side of it). All half-edges associated with a face are counter-clockwise which can be obtained by traversing its representative Halfedge. To reach the other face, we can go to the twin of the half-edge and then traverse the other face. Each half-edge also has a pointer to its origin vertex which can be used to get all the Vertex points of the corresponding face.

Results

This algorithm to construct a DCEL takes time complexity of O(V+E). From the analysis of example dataset runtimes, we can easily see that increase in number of vertices and edges directly leads to increase in DCEL construction time.



The output of the algorithm is direct traversal of representative edges of all faces present in the input. In the above example of Hyderabad road network dataset, it shows all the faces bounded by the roads. All faces might not be visible directly as the road system branches without forming many faces. But as we zoom, the faces can be spotted between roads.

Conclusion

The time complexity O(V+E) cannot be reduced further because to store the graph, it is required to traverse through all Vertices and Edges present in the graph. Although the current implementation only supports connected graphs, it can also be extended to store disconnected components using techniques like hidden vertices.

From the above results, we can be sure that DCEL can be a good candidate data structure for easy representation of planar graphs. But as every other data structure, even DCEL has it downfalls. The main one being the amount of Storage required to store. The storage requirement can be further reduced if we only need to store Vertices or Edges, but this does not give huge improvements. Also, as the size of DCEL increases, the time to add an extra edge increases in a linear fashion which might be bad based on the application.DCEL is a simple data structure which helps to easily manupulate the graph structre, but it might not be an appropriate choice for low memory applications such as in embedded systems.

Polygon Triangulation

Introduction

The decomposition of a polygon into triangles whose union again results in the original polygon is called polygon triangulation. In the method, no new Vertices are added. Only the diagonals of the polygon results in the triangulation. There need not be a unique triangulation for a given polygon.

There are various methods which can be used to perform polygon triangulation. In our implementation, we discuss two types of triangulation which are: 1) Ear Clipping Triangulation. 2) Plane sweep monotone division combined with Monotone triangulation.

How to Run

The src folder contains the source code for the convex hull program. `g++` from the GNU compiler suite is required to compile the program to a executable.

Steps to Compile:

1) `cd` into the src directory
2) Run `g++ main.cpp` which generates an executable called `a.out` in the same directory
3) Run the executable using `./a.out` (on linux)
    1) The executable takes a dataset from command line argument. For example, to use an existing dataset, run `./a.out ../datasets/complex.txt`
    2) If no command-line argument is given, it takes input from the shell directly (stdin)

Input

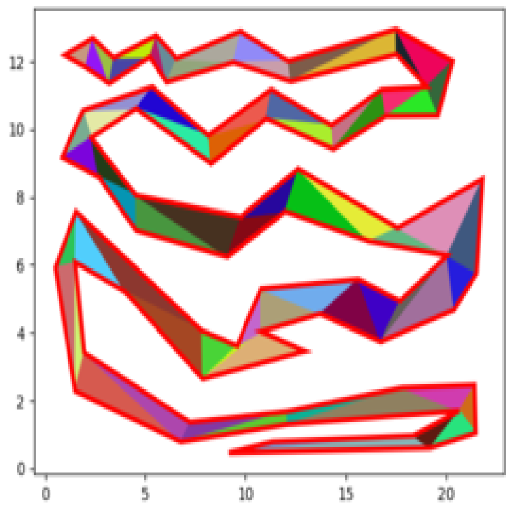
The required file format for the algorithm to work correctly is:

- The input **must** be a clockwise ordering of Poins present on the polygon.
- First line must contain the no of Points to be taken as input by the program.
- Each of next line must contain 2 integers, space seperated denoting the (x, y) coordinates of each point.
- Each coordinate must be of integer type in the range -10^8 to 10^8.
- No of coordinates must be less than 1 Billion.

Output

Each line of output contains three coordinates of points present on each triangulation.
The last two lines of output show the time taken to take input in microseconds and the time taken by the algorithm to compute Triangulation (also in microseconds).

This is the output of one of the datasets long.txt



Documentation and Report

Documentation of this algorithm, functions and classes can be found in the `docs` folder in the current directory. Open the index.html file from the docs directory with your preferred browser to go through the documentation

Performance Analysis

Analysis is performed with a system running (does not include printing time):

- OS: Arch Linux (64Bit) running Linux Kernel version 5.7.2
- Processor: Intel Core i7 7700HQ
- RAM: 8GB
- Compiler: GNU G++ (GCC) 10.1.0

The following observations are recorded:

Time taken to triangulate basic shapes:

Filename	Input Vertices	Computed Triangles	Line Sweep Triangulation Runtime	Ear Clipping Triangulation Runtime
triangle.txt	3	1	30 microsec	16 microsec
downConvex.txt	6	4	43 microsec	23 microsec
test.txt	6	4	35 microsec	24 microsec
square.txt	4	2	36 microsec	18 microsec
hexagon.txt	6	4	38 microsec	16 microsec
monotone.txt	15	13	109 microsec	75 microsec
uniMonotone.txt	8	6	115 microsec	42 microsec
complex.txt	17	15	188 microsec	17 microsec
strange.txt	16	14	148 microsec	122 microsec
star.txt	10	8	159 microsec	101 microsec
spiral.txt	32	30	278 microsec	214 microsec
tank.txt	55	53	429 microsec	452 microsec
long.txt	72	70	593 microsec	1668 microsec

Algorithm Approach

Ear Clipping

According to the two ears theorem, any simple polygon with minimum of 4 vertices without holes has atleast two “ears”. An Ear is defined at the vertex where internal angle is less than PI and the line joining the adjacent vertices is a diagonal of the polygon.

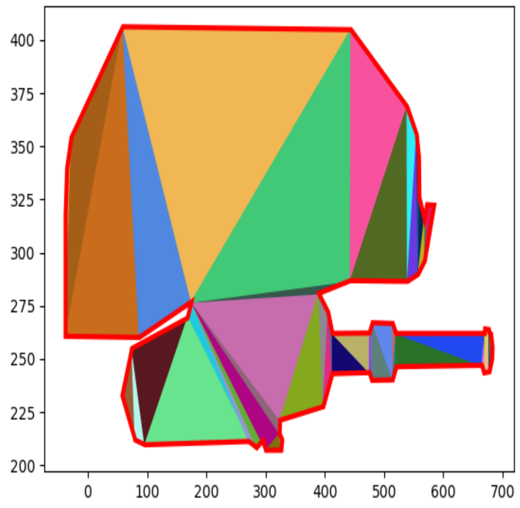
The algorithm works by removing away these “ears” (which are triangles) until the complete polygon is triangulated. This algorithm is easy to implement, but slower than some other algorithms, and it only works on polygons without holes. The runtime complexity of this algorithm is O(n^2)

Plane Sweep Monotone Triangulation

A monotone polygon can be easily triangulated in O(n) complexity. A simple polygon is said to be monotone w.r.t a line L only if any line perpendicular to L passes through the polygon at most twice. Any monotone polygon can be divided into two monotone chains. A polygon that is monotone w.r.t the x-axis is called x-monotone. Given x-monotone polygon, the greedy algorithm begins by walking on one chain of the polygon from top to bottom while adding diagonals (forming triangles) whenever it is possible.

If a polygon is not monotone, it can be partitioned into monotone sub polygons in O(n log n) time using line/plane sweep method. Generally, this algorithm can triangulate a planar subdivision with in O(n log n) time using O(n) space.

Results



In the above image, the red boundary represents the input given to the algorithm and the random colored triangle represent the output given by the program.

It can be observed from the examples that increase in number of vertices affects the runtime of ear clipping triangulation more than the plane sweep monotone triangulation. From the results of the above datasets, we can see that for just an increase of vertices from 50 to 70 causes the ear clipping algorithm to triple its triangulation time.

Conclusion

From the above comparisions, we can see that ear clipping algorithm, even after being a simple algorithm, due it is asymptotic complexity being O(n^2), it surely is not recomended for applications where number of vertices go beyond 70 (which almost always happens).

If we know that the input only has a monotone polygon, it is even more easier and faster to triangulate it in just O(n). The main plane sweep algorithm tries to use this to its advantage by dividing the polygon first into monotones.