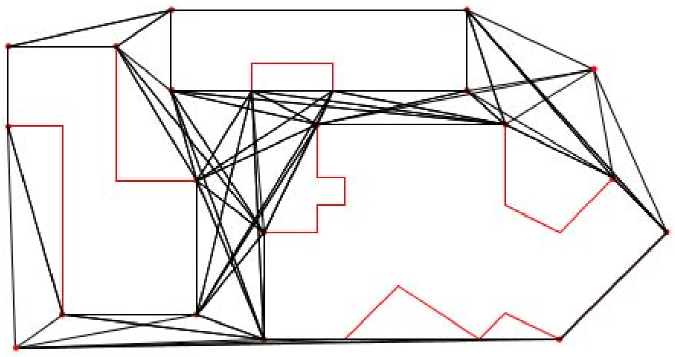
Convex Hulls and Path Finding

Robert McConnell

Design of Algorithm

Visibility Graph: [*VisibilityGraph.java, generatePaths()*]
'A *Visibility Graph* is a *graph* of intervisible locations, typically for a set of points and obstacles in the *Euclidean plane*. Each *node* in the graph represents a point location, and each *edge* represents a *visible connection* between them.'

Once the convex hull of each polygon on the map was found *(described in the Implementation Design section below)* I wanted to create a *medium* for a *shortest path* algorithm such as *Dijkstra's* or *A** to run. Initially I thought of creating a large grid with the polygons being set to *null* and running the algorithm over that but I did not think that it would be *computationally efficient* (or *memory efficient*), also the route would only be as optimal as the size I made the cells (the smaller the better). I knew that in *2D Euclidean Space* the shortest path between two points is a straight line so I decided to use a visibility graph.



Visibility Graph showing the 'Lines of Sight' between 3 Polygons, a Start Point and an End Point. Black Lines are Visibility Graph 'Edges', the Red Points are the Visibility Graph 'Vertices' and the Red Lines are the Polygons.
[TEST-MAP-1.TXT]

A visibility graph is a *graph data structure* that contains the vertex of each point of each polygon and the lines between each of the vertices that can *'see'* each other. i.e if a line between two vertices does not cut through a polygon it is a possible path and should be added to the visibility graph. The final visibility graph will contain *every* possible route between the start and end point and then can be used with a shortest path algorithm to find the *optimal route*.

To use a *path finding algorithm* such as *Dijkstra's* (discussed below) it is required that *weights* between graph *nodes* (vertices) are defined. For the weighting I used the distance between vertices. To calculate the distances I wrote a simple *dist()* method that used the *Euclidian distance formula*. When building the graph I added the *edges* between vertices and the distance between them to a *Graph.Edge array list*. As I would be adding to the graph in several locations it was beneficial for me to use an

array list over an array as it has the benefit of being able to add items without requiring an index. I felt that this pro outweighed any cons with converting it back to an array.
To add a new edge to the graph I would check for *intersections* and if there were none I could add using:

paths.add(new Graph.Edge(vertex1, vertex2, dist(vertex1, vertex2);

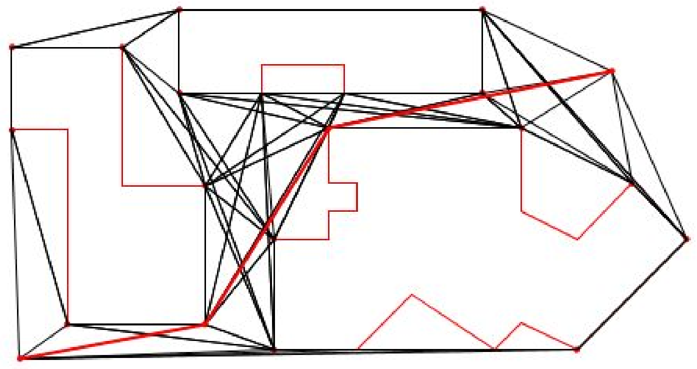
Paths are added to the paths list in several parts of the *parent for loop*. I have a loop that adds a line between each vertex on each polygon that does not intersect an edge and another one for adding links between the start and end points. The for loop structure looks scarier than it is, to make the code more easy to read and understand I broke it up a bit, so even though it looks like there are 2 for loops for iterating over the points, it is a for loop for going through each polygon and then each point in each polygon. This is the equivalent to having one for loop iterating over an array containing all of the points.

The intersections are checked using a method I wrote named *lineOfSight()* that takes in two *vectors* as its parameters and returns a *boolean* depending on if there is an intersection or not. Whenever it returns *true* in one of the for loops it increments an *'intersections'* variable and if this variable is equal to 0 after checking all of the edges for intersections a path may be added to the paths array list. I intended to use this to exit the for loop early if an intersection was found but I did not have time to return to it.

Complexity: Implementing the visibility graph was quite computationally expensive. The method I used involved comparing every point against every other point O(N^2) where N is every vertex on every polygon, and determining if they intersected any edge for a combined complexity of O(N^3). I know that there are possible implementations with a better complexity, the best I saw was one that used arrangements and had a complexity of O(N^2).

Improvements: As I mentioned I would like to *optimise* the for loops but apart from that I think that I would like to look at a more efficient implementation of a visibility graph with a *lower complexity*.

Dijkstra's Algorithm: [*Dijkstra, dijkstra()*]
'Djikstra's algorithm solves the problem of finding the shortest path from a point in a graph (the source) to a destination. It turns out that one can find the shortest paths from a given source to all points in a graph in the same time, hence this problem is sometimes called the single-source shortest paths problem.'



Shortest Path between 3 Polygons from a Start Point to an End Point calculated using Dijkstra's Algorithm. Black Lines are Visibility Graph 'Edges', the Red Points are the Visibility Graph 'Vertices', the thin Red Lines are the Polygons and the thick Red Line is the Shortest Path. [TEST-MAP-1.TXT]

The shortest path algorithm that I decided to use is called Dijkstra's Algorithm. It is an algorithm for finding the shortest path between nodes in a graph (in this case the visibility graph). Each of the nodes are separated by weighted links. I have the weighting to equal the distance between each node as explained above. A large portion of the visibility graph is actually within Dijkstra.java due to them sharing the graph data structure.

Each vertex in the algorithm records several pieces of information:

- the shortest distance between current node and the starting node.
- the previous node in this shortest path.
- a way of indicating if the node has been checked.

Initially the shortest distance to each node is set to a very high number (*Double.MAX_VALUE*, i.e infinity) and all of the nodes are marked as unchecked. The algorithm will then process the vertices one by one and the processed vertices will receive a shortest distance value. When all of the vertices have been processed a shortest path will be present.

When finding new paths then algorithm will examine every vertex that has not been marked as checked. For each vertex a new path from the starting point to the vertex is found. If the path from the vertex to the starting point is shorter than the current shortest value then this is the length of the new shortest path. (will be initially as the initialised value is infinity)

The act of defining a new shorter path is known as 'Relaxation', this is the code that I have for it:

```
final double alternateDist = u.dist + a.getValue();
if (alternateDist < v.dist) { // shorter path to neighbour found
        q.remove(v);
        v.dist = alternateDist;
        v.previous = u;
        q.add(v);
}
```

In this code snipped the alternateDist is made by combining the the path from the starting point to u and the edge from u to v. If it is less than the current shortest distance then the old path is replaced with the new shorter path.

Vertices are stored in a TreeSet which provides the implementation of a set and uses a self-balancing binary search tree for storage. Access and retrieval times are fast which makes it a good choice for storing large amounts of sorted information that must be found quickly.

Complexity: The graph is built from a set of edges with a complexity of O(E log V). The complexity of removing from a TreeSet is O(log N).

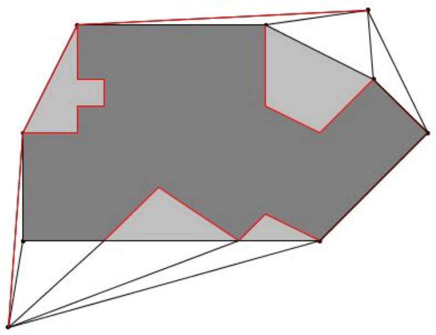
Improvements: The code ended up being messier and less modular than I would have liked, I think it could have been structured better as I had difficulty accessing different elements of it and returning the path as an array etc. I also would have liked to have added a heuristic element and implemented the *A* path finding algorithm* but once Dijkstra's was working I was happy with the result.

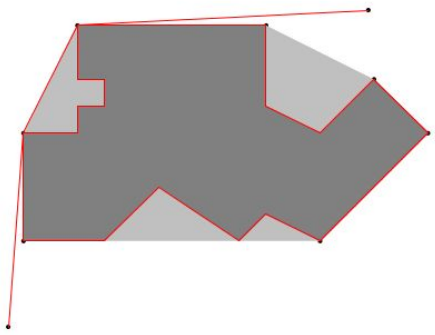
The combination of the Visibility Graph and Dijkstra's Algorithm should always provide an optimal solution (there is a bug in the convex hull implementation that I have mentioned below which could affect the outcome of the path but this is not due to a problem with the visibility graph or Dijkstra's algorithm).

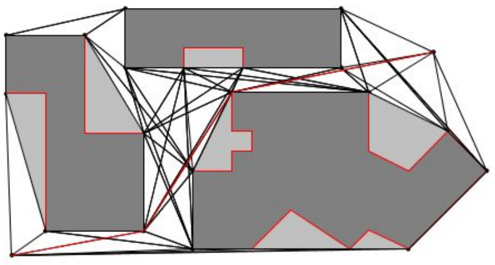
The visibility graph is based on some geometry properties that ensure that the shortest path will be one of the paths in the graph:

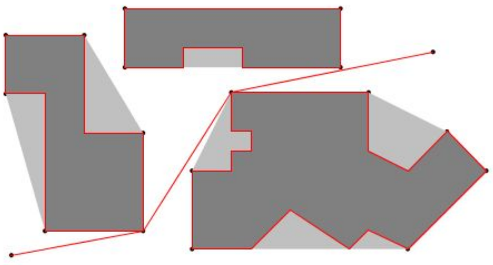
- *In two dimensional Euclidean planes, the shortest trajectory between two points is a straight line.*
- *The length of a sequence of straight line trajectories is the sum of the lengths of each line.*
- *Since a polygon is made up of straight lines (or edges), the shortest path which completely circumnavigates a convex polygon contains all of the polygon's edges.*
- *To circumnavigate a non-convex polygon, the shortest path contains all of the edges of the polygons' convex hull.*
- *A point exactly on the edge of a polygon collides with that polygon.*

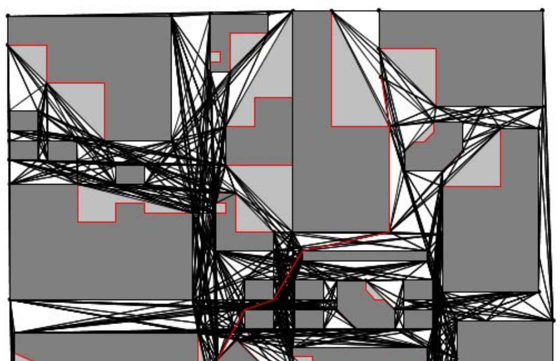
On the next page I have attached figures of the outputs for the test maps provided on loop.

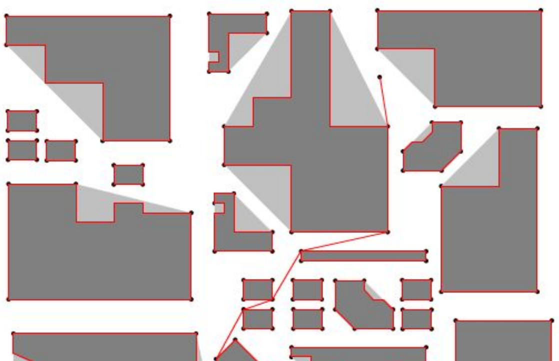

TEST-MAP-0.TXT -v


TEST-MAP-0.TXT


TEST-MAP-1.TXT -v


TEST-MAP-1.TXT

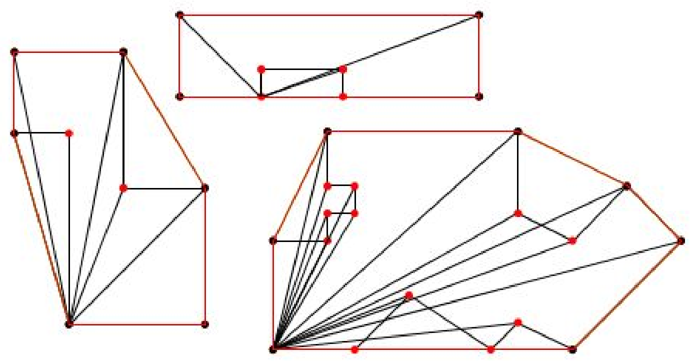

TEST-MAP-2.TXT -v


TEST-MAP-2.TXT

Implementation Design:

Convex Hull: [*GrahamScan.java, findConvexHull()*]
'A set of points is a *convex set* if a straight line segment, joining any pair of points in the set, lies entirely inside the set. The *convex hull* of a set *P* is the smallest convex set containing all points in *P*.'

The first part of this assignment was to find the convex hull of every polygon on a provided map. This would ensure that the rest of the path finding procedure runs as efficiently as possible as the shortest path around an object will always be around its convex hull.



Convex Hulls of 3 Polygons, Black Points are Hull Points, Red Points are Discarded Points, Red Lines are Edges between Hull Points and Black Lines show the Points sorted by Polar Angle. [TEST-MAP-1.TXT]

I used the *Graham Scan* algorithm for this as it is the most efficient method of determining convex hulls. The algorithm works as follows:

I. Find the *lowest point* (minimum y-coordinate)
II. Sort all of the ports by decreasing *polar angle* with respect to the lowest point.
III. Initialise the *hull points* with the first two sorted points (guranteed to be on the hull)
IV. Iterate through the points and determine the direction of any turns that are made. If a right turn is made, add to the stack and continue. If a left turn is made backtrack and remove from stack.

Before determining the convex hull I *pre-processed* the data to reduce the amount of points the Graham Scan algorithm would have to process. This was done by removing any points that are within a *quadrilateral* formed by the *max* and *min*, *x* and *y* coordinates. *(I am not sure if my implementation was actually more efficient than not pre-processing the points).*

After the Graham Scan algorithm returned the hull points the collinear points need to be removed as the convex hull is defined as the *minimal* set of points enclosing *P*.
This was done by finding the area of a triangle that is made using 3 points. If the area is 0 then the points are collinear. Due to some rounding errors I needed to set a threshold of 0.001 as some areas were not equaling 0.

Graham Scan Implementation:

I. To find the lowest point I made two simple methods, *lowestPoint()* and *swapLowest().* Together these methods determined the lowest point without sorting the points and disrupting the order, it is also more efficient than a sorting algorithm. It worked by setting a value as the minimum and replacing it if a lower value was found.
II. To sort the points by polar angle I wanted to implement the *generic interface Comparable* so that I could use Java's built in sorting methods (*Arrays.sort* method in *java.util.Arrays*). To do this I needed to write a *compareTo()* method, I had a problem in that I wanted to use the cross product method of determining the polar angle but the *compareTo()* method can only receive 2 variables (the *instantiated* one and the *parameterised* one). To solve this problem I created vectors (*Vector.java*) between each point on the polygon (i.e each edge) and sorted them using a *merge sort* algorithm that I wrote (also works with *Arrays.sort*).
III. I created a stack (*HullStack.java*) that I found to be the best data structure for Graham Scan. To add the two initial hull points I simply had to push them onto the stack.
IV. The method for determining if a right or left is made is similar to the cross product used for sorting by polar angle. The method that I created is called *turningDirection()* and it is located in *Point2D.java*.

Complexity: Graham Scan's complexity is dominated by the complexity of the sorting algorithm used. As I used *merge sort* the complexity of finding the convex hulls is O(N log N). (When using *Arrays.sort* the *quicksort* sorting algorithm is used which has the same complexity as *merge sort*)

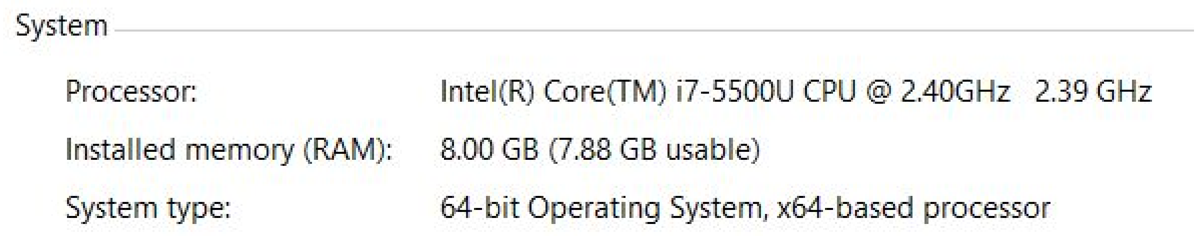
Improvements: There is currently a bug in my code that I was unable to resolve before submission. It occurs in one polygon in *TEST-MAP-2.TXT* (and presumably other possible maps). I have tried several solutions to fix the problem but none of them worked. I believe that it has something to do with the fact that I used the *compareTo()* interface to sort the points and it only sorts points by polar angle but does not have a way of sorting points that have equal polar angles (the problem occurs in a polygon where there are 4 points on the same x-coordinate). This is just a theory as I have not investigated the problem enough. I was surprised that it was not caught by the Graham Scan algorithm and the pre-processing of the data.

I tried to make the code as modular as possible with OOP in mind. I have a .java file called *pathFinder.java* where I tie all of the parts of the program together. It imports the maps using *ShapeMap.java* and *MapFileReader.java*. It then generates the convex hulls of each of the polygons by calling on the method in *GrahamScan.java*. It then creates the visibility graph using *VisibilityGraph.java* and gets the shortest path using *Djkstra.java*. It also checks for command line arguments, prints the visualisations and records the distance and run times.

Throughout the code files I have listed above I use several data structures that I thought were relevant in their usage. I use *HullStack.java* for the Graham Scan stack, I also use *Point2D.java*, *Polygon2D.java* and a *Vector.java* class.

Algorithm Analysis:

Each aspect of this program has a different *computational complexity* and contributes differently to the overall complexity of the program. Below I have made a table comparing the *input size* (number of points) and the *execution time*. A large portion of the execution time is due to *StdDraw* but I decided to leave the animations as it was consistent across all of the maps so shouldn't provide an unfair advantage or disadvantage. I have also run the code on each map file 3 times and taken the average to avoid CPU/RAM spikes.

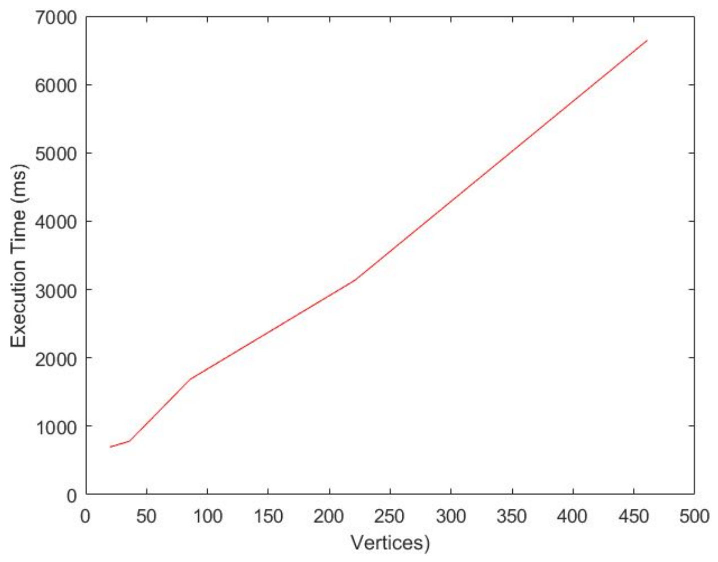


System

Processor:	Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz 2.39 GHz
Installed memory (RAM):	8.00 GB (7.88 GB usable)
System type:	64-bit Operating System, x64-based processor

This is the specifications of the laptop I am executing the code on.

MAP	VERTICES	EXECUTION TIME
TEST-MAP-0.TXT	20	697
TEST-MAP-1.TXT	36	805
TEST-MAP-2.TXT	221	3136
DEMO-MAP-1.TXT	36	757
DEMO-MAP-3.TXT	86	1689
DEMO-MAP-2.TXT	461	6647



Plot of the Results Based on Loop Map Files

Complexity of Different Aspects of Program:
Convex Hulls: O(N log N)
Visibility Graph: O(N^3)
Dijkstra's Algorithm: O(E log V)

Total Complexity:
As the Visibility Graph is *'bottlenecking'* the complexity the time complexity the program is O(N^3).

Looking at the plot it seems like the algorithm is O(N log N). This does not match the derived computational complexity but with a small number of test files and relatively small values of N it is expected that the results would not exactly display the theoretical complexity. Also with a small input size the execution time wildly varies with each execution.