Student Declaration of Academic Integrity

This form **must** be filled in and completed by the student submitting an assignment.
Assignments submitted without the completed form will not be accepted.

Name: **Ciaran O'Donnell**
Student ID Number: **15414048**
Programme: **ECE**
Module Code: **EE324**
Assignment Title: **Project 1, Convex Hulls and Path Finding**
Submission Date: **06-12-2017**

- I understand that the University regards breaches of academic integrity and plagiarism as grave and serious.
- I have read and understood the DCU Academic Integrity and Plagiarism Policy. I accept the penalties that may be imposed should I engage in practice or practices that breach this policy.
- I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations, paraphrasing, discussion of ideas from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the sources cited are identified in the assignment references.
- I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.
- I have used the DCU library referencing guidelines (available at: http://www.library.dcu.ie/LibraryGuides/Citing&ReferencingGuide/player.html) and/or the appropriate referencing system recommended in the assignment guidelines and/or programme documentation.
- By signing this form or by submitting this material online I confirm that this assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.
- By signing this form or by submitting material for assessment online I confirm that I have read and understood DCU Academic Integrity and Plagiarism Policy (available at: http://www.dcu.ie/registry/examinations/index.shtml)

Name:_____Ciaran O'Donnell_____

Date:_____06-12-2017_____

EE324 PROJECT 1

Convex Hulls and Path Finding

Algorithm Input Setup:

So the first thing I did for this project was to download the supplied code and set it up so that the map files could be read and the data saved in memory in a format suitable for this algorithm. I downloaded the java files ObstacleMap, MapFileReader, Polygon2D, and Point2D. ObstacleMap stored the data of the map input. It created a Polygon2D object of the obstacle to be avoided and a Point2D object of the source and the destination. Polygon2D stored an array of Point2D objects that make up said polygon. MapFileReader is a class used to read the data of the input map. ObstacleMap, Polygon2D, and Point2D all had a number of useful functions.

Jarvis March Implementation:

The next step was to use the classes given to compute the complex hull of the obstacle. I used the Jarvis march algorithm to complete this. I started by implementing a function that would take all the points and get an initial point to start at. This point would need to be on the convex hull. The only points that are certain to be on the convex hull are points at the extremities of the axis, i.e. the leftmost, rightmost, highest, or lowest. I found the lowest point by comparing the Y value of all points of the polygon. I had to make sure my code worked in the case of collinear points. I did this by using a list. If the point was lower than my current lowest it would reinitialize the list and add the new lowest point to it, however if a point was the same it would simply add it to the list. It would then find the leftmost point in the list by comparing there x values and removing the larger one until the list had only one element. This element would be on the convex hull.

```
For i = 0 : all points in array:
        If ary[i].Y < LowestList(0).Y
                Reinitialize LowestList(0)
                Add ary[i]

        If ary[i].Y == LowestList(0).Y
                Add ary[i]

While LowestList.size < 1
        If LowestList(0).X < LowestList(1).X
                Remove LowestList(1)
        Else
                Remove LowestList(0)
```

The Jarvis March code works by finding the most counter-clockwise point and adding it to the list of hull points until back at the initial point. This means that I needed a function to find the most counter-clockwise point to the current one. I did this by comparing the cross product of two points to the current point and using the value to determine which is counter clockwise.

P0 = source point
P1 and P2 are points tested

K=cross product of P0 P1 and P0 P2

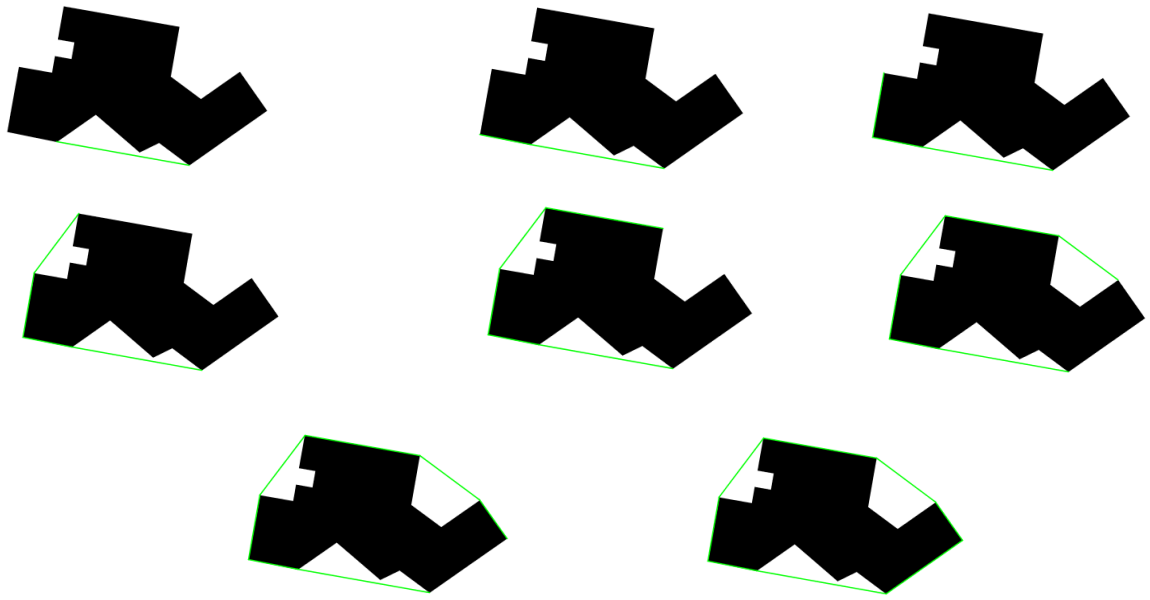
If k>0 P1 is counter clockwise
If k<0 P1 is not counter clockwise

However, in some cases the two points being tested can be at the same angle. In this case I simply tested their distances and used the further one as this would simply leave the other point lying on the convex hull but not included in the list of hull points which was fine for my implementation and saved on computation time as it wouldn't add unnecessary points.

If K==0 return furthest

The final step in my Jarvis March class was using these two functions to find the convex hull. So, I got the initial point as described, then went through all the polygon points adding the most counter clockwise point each time until the point added is equal to the initial point.

```
InitialPoint                //gotten with function
CandidatePoint    //Random Point in Polygon
HullList                    //List for hull points to be stored

Do {
        For i = 0 : Amount of Points
                If (Point(i) is counter clockwise of CandidatePoint from HullList(HullList.size))
                        CandidatePoint = Point(i)
                End
        End
        HullList.add(Candidate)
} While (CandidatePoint != InitialPoint)
```



Path Finding Algorithm:

So now with the ability to read and store the input data and the ability to compute the convex hull it was time to develop my own algorithm to find a path from the source to the destination avoiding the obstacle. I decided to write the majority of this algorithm in a new class I named FindPath. I first took time to decide how I would go about doing this. I decided to start I would use a nearest neighbor technique to get from the source and destination points to the hull.

Nearest Neighbor Path to Hull:

So, I knew I would, in order to use a nearest neighbor technique to get to the hull, need to be able to test three things, the distance between points, if a line from one point to another intersects any other, and if a point was on the hull. I decided to add these as their own functions.

I first added a function called Distance to the Point2D class that would take another Point2D as an argument and return the distance between them as a double.

```
return sqrt[(X1*X2)^2 + (Y1*Y2)^2]
```

I then added a function to determine whether a point was on the convex hull. I did this by simply searching through all the points on the Hull.

```
For i = 0 : hull.size
        If (hull(i) == point) return true
        EndFor
Return false
```

It's also noteworthy to mention this code could be edited to search for a point in any polygon and not just the hull however in this project I did not need to.

The next step was to write a function to test if a line intersects with any line of the obstacle. I used the Java geom.Line2D library to do this. Luckily the way the MapFiles we were using are configured the points of the obstacle are always going to be sorted in our case. This means we can test for intersections simply by creating the line we want to test and then testing it against all the lines of the polygon by stepping up through them.

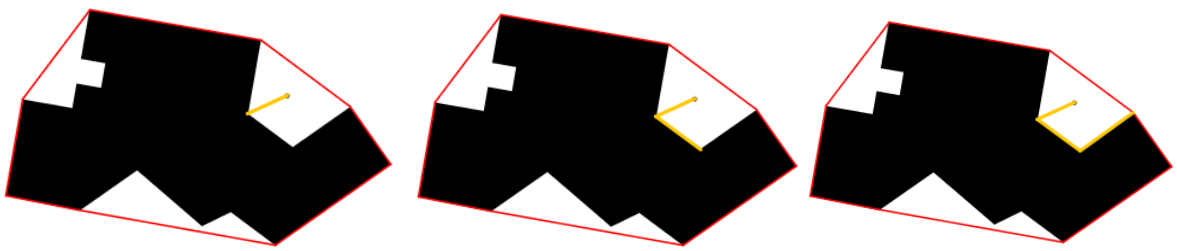
```
MyLine2D = Line2D(p1, p2)
For (i = 0 : amount of obstacle points)
        P3 = ObstaclePoint(i)
        P4 = ObstaclePoint(i+1)
        TestLine = Line2D(P3,P4)
        If ((P1!=P3 && P1!=P4) && (P2!=P3 && P2!=P4))
                If (MyLine.intersects(TestLine)) return true
endFor
return false
```

The first if statement is necessary to avoid false positives when the line ends at the point of another line.

So, with these functions created I can finally write the code to use nearest neighbor to find my way from a point to the hull for use with the source point and the destination point. To find the way out to the hull using nearest neighbor, while the current point is not on the hull search through all obstacle points and save the nearest one to a temporary variable. Then test this point to see if it intersects any point. If it doesn't add it to your route to the hull, if it does add it to a list of unreachable points and remove it from the obstacle points to be tested. Once a new point is found and added to the route all unavailable points are placed back into the obstacle test points and the point found is removed. Continue this until the point found is on the hull.

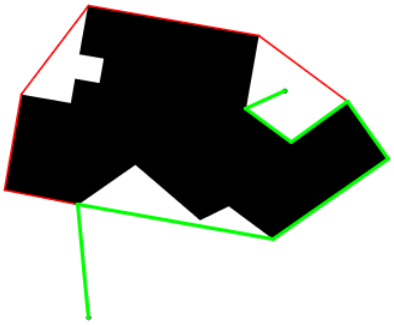
```
ListPoints = obstacle points
ListUnreachable = empty
ListPath = empty
P = current point

While (P is not on hull)
        ShortestDistance = Infinity
        For (i = 0 : ListPoints)                                    //this for loop finds closest point
                If (P DistanceTo ListPoints(i) < ShortestDistance)
                        ShortestDistance = P DistanceTo ListPoints(i)
                        Index = i
                endIf
        endFor

        If (P to ListPoints(index) does not intersect)      //if the point doesn't intersect-
                ListPath.add(ListPoints(index))             //it's added to the path
                P = ListPoints(index)
                ListPoints.remove(ListPoints(index))
                ListPoints.addall(Intersections)            //adds back in previously-
        endIf                                                       //ineligible points
        Else
                Intersections.add(ListPoints(index))                //if it intersects its added to-
                List.remove(index)                                  //ineligible points
        endElse
```

This will give a route from any point not in the object to the convex hull of the objest without intersecting the object. This method may not always find the shortest path to the convex hull but it will be pretty accurate and I later implement another function that improves its accuracy further.



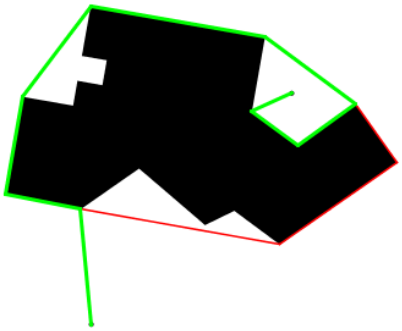
Full Path

The next step was to join up the two hull points that my source and destination found a path too. The way I decided to do this was to step through the hull until I came across the point that the nearest neighbor method reached from my source point. Once found I would add each hull point incrementally until I got to the hull point that my destination path reached. I then added the points from that path into the path. I end with a path from the source to the destination however it is not a very short.



```
Polygon2D sourcePath = Path from source to hull
Polygon2D destPath = Path from destination to hull
Int i = 0
While (hull(i) != sourcePath(sourcePath.size()-1)       i++ //gets to the hull point of SourcePath
i++
While (hull(i) != destPath(destPath.size()-1)
          sourcePath.add(hull(i))
          i++
if(i==hull.size()) i=0                                  //wrapped fully around hull
endWhile

for (i=destPath.size-1 : 0)
          sourcePath.add(destPath(i))
endFor
```

This can also then be done going the other way around the hull simply by starting with i=hull.size()-1 and decrementing down to zero. I did both directions and stored them for later use in finding the shortest path.



As you can see this path is way longer than necessary so my next step was to optimize the paths found.

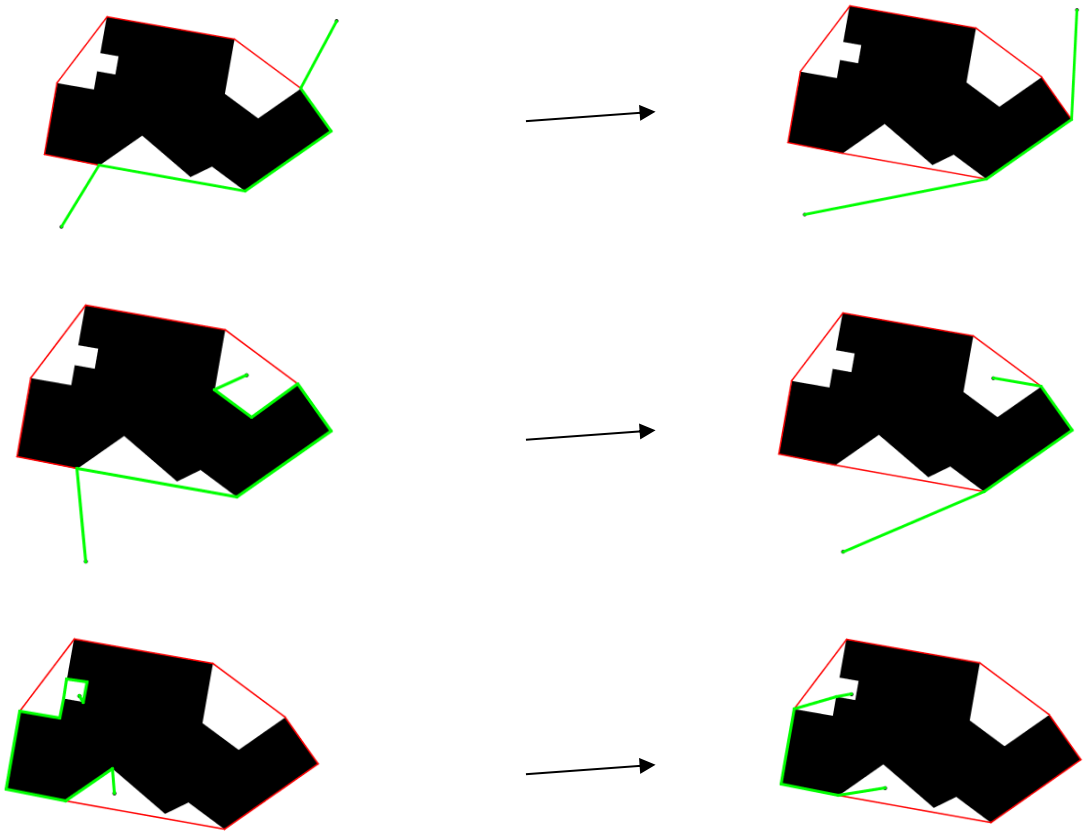
Optimization of Path

So I know had a Path that got from the source to the destination but it is a lot longer than is necessary. So I realized the path can be drastically shortened and even shortened to the shortest possible route in most situations if from each point it travels to the furthest possible point on the path without intersecting. Therefore, I wrote a function that travels along the path from a point testing each point to see if it intersects. When it finds a point it intersects with it adds the previous point to its new path and works on from this new point. For this reason a stack is a very suitable data type for this function. However, the path around the convex hull is already the most efficient path. So I ran this to get the most efficient path from the source to the hull and from the destination to the hull only.

```
CurrentPath
Stack SourcetoHull
Stack DesttoHull
SourcetoHull.push(CurrentPath(0));                    //sets to source point
DesttoHull.push(CurrentPath(CurrentPath.size-1))      //sets to dest point
Int i =1

While (SourcetoHull.peek is not on hull)              //using earlier function
        If (SourcetoHull.peek to CurrentPath(i) does intersect)
                SourcetoHull.push CurrentPath(i-1)    //add point before (becomes new working point)
        EndIf
        I++
EndWhile
                                                      //do the same but in reverse for destination point
Int k = CurrentPath.size-2                            //starting from second last point (point before dest)
While (SourcetoDest.peek is not on hull)              //using earlier function
        If (SourcetoDest.peek to CurrentPath(i) does intersect)
                SourcetoDest.push CurrentPath(k+1)
        EndIf
        k--
EndWhile

i--                                                   //need i to add in rest of hull points
FinalPath = SourcetoHull                  //Simple Code to Transfer Path stack into polygon here

While (FinalPath(FinalPath.size) != DesttoHull.peek)  //adds hull points until it reaches the point the
        FinalPath.add(CurrentPath(i)                  //destination got to
        i++
endWhile

While (!DesttoHull.empty)
        FinalPath.add(DesttoHull.pop)                 //empties DesttoHull into FinalPath
endWhile
```

We are then finally left with a path that has its distance shortened significantly. This function drastically lowers the inaccuracy of the nearest neighbor function used earlier. In my code I did this for the two routes I got earlier, measured the distance of both and returned the shorter one. This means the code will always find the shorter route around the obstacle bar a few exceptions.

Here is the difference the optimization makes:



So when we do this for both directions around the hull and return the shorter path each time we are left with a path that, is or is close to, the shortest possible path.

Computational Efficiency

So we no that the time-complexity of Jarvis March is on average O(n+h) but seeing as it is only ran once at the start of my algorithm it can be ignored for the overall time complexity. Getting from the route to Hull using nearest neighbor is O((n^2+u*n)*k) where n is the number of points (squared as it must check all points for intersections) u is the number of unreachable points (multiplied by n as it also checks for intersections) and k is the amount of points traversed to reach the hull. Getting the route around the outside of the hull is simply O(s+h+d) where s is the source route, h the hull, and d the destination. Optimizing the route is just of O(s*n + h + d*n) s is the source path h is the hull points it must traverse and d the destination path, s and d multiply n as they must test for intersections each step. So my overall time complexity is O((n+h)+((n^2)+(u*n))*k+s+h+d+sn+h+dn)) breaking this down by eliminating lower factor elements gets O((n^2+un)*k+sn+dn)) the highest order element is O(kn^2) so this is the time complexity of the overall algorithm, although its worth keeping in mind all the smaller order elements that will have an impact to the overall time to compute.

Appendix:

The only change I made to the supplied code was to add a distance method to Point2D:

```
public double Distance(Point2D p)
                return Math.sqrt((p.getX()-x)*(p.getX()-x) + (p.getY()-y)*(p.getY()-y));
        }
```

Here is my JarvisMarch class:

```
package Navigator;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;

import edu.princeton.cs.introcs.StdDraw;

public class JarvisMarch {


        public static Polygon2D findConvexHull(Polygon2D polygon) {

                Point2D init = getInitialPoint(polygon);
                Point2D candidate = polygon.getIndex(0);
                List<Point2D> hull = new ArrayList<Point2D>();
                hull.add(init);

                do {
                        for(int x=0; x<polygon.size(); x++) {
                                if(isCounterClockwise(hull.get(hull.size()-1), candidate,
                                                                polygon.getIndex(x))) {
                                candidate = polygon.getIndex(x);
                                }
                        }
                        hull.add(candidate);
                }while (candidate.equals(init) != true);

                Polygon2D hullpoly = new Polygon2D();
                for(int j=0; j<hull.size(); j++) {
                        hullpoly.addPoint(hull.get(j));
                }
                return hullpoly;
        }

        public static Point2D getInitialPoint(Polygon2D polygon) {
                Point2D ary[] = polygon.asPointsArray();
                List<Point2D> initlis = new ArrayList<Point2D>();
                Point2D init = new Point2D(Double.POSITIVE_INFINITY, Double.POSITIVE_INFINITY);
                initlis.add(init);

                for(int i=0; i<polygon.size(); i++) {
                        if(ary[i].getY()<initlis.get(0).getY()) {
                                initlis = new ArrayList<Point2D>();
                                initlis.add(ary[i]);
                        }
                        if(ary[i].getY()==initlis.get(0).getY()) {
                                initlis.add(ary[i]);
                        }
                        while(initlis.size()>1) {
                                if(initlis.get(0).getX()<initlis.get(1).getX()) {
                                        initlis.remove(1);
                                }
                                else {
                                        initlis.remove(0);

                                }
                        }

                }
                init = initlis.get(0);
                return init;
        }

        public static boolean isCounterClockwise(Point2D p1, Point2D p2, Point2D p3) {

                double x0 = p1.getX();
```

```
                double y0 = p1.getY();
                double x1 = p2.getX();
                double y1 = p2.getY();
                double x2 = p3.getX();
                double y2 = p3.getY();

                double k = (x1-x0)*(y2-y0)-(y1-y0)*(x2-x0);

                if(k>0) return true;
                else if(k<0) return false;


                //following runs if colinear or is the same point
                double dist1 = (x1 - x0)*(x1 - x0) + (y1-y0)*(y1-y0);
                double dist2 = (x2 - x0)*(x2 - x0) + (y2-y0)*(y2-y0);

                if(dist2>dist1) return true;
                return false;
        }

        public static void main(String args[]) {



        }

}
```

And my Path finding algorithm:

```
package Navigator;

import java.awt.Color;
import java.awt.geom.Line2D;
import java.util.*;

import edu.princeton.cs.introcs.StdDraw;

public class FindPath {

        private static ObstacleMap points;
        private static Polygon2D obstacle;
        private static Polygon2D hull;
        private static boolean v = false;
        private static int delay = 500;

        private static boolean IsOnHull(Point2D p) {
                for(int i=0; i<hull.size(); i++) {
                        if(hull.getIndex(i).equals(p)) return true; //tests if point is on hull
                }
                return false;
        }

        private static boolean Intersects(Point2D p1, Point2D p2) {
                Line2D line1 = new Line2D.Double(p1.getX(), p1.getY(), p2.getX(), p2.getY());
                for(int i=0; i<obstacle.size()-1; i++) {
                        Point2D p3 = obstacle.getIndex(i);
                        Point2D p4 = obstacle.getIndex(i+1);
                        if((!p1.equals(p3) && !p1.equals(p4)) && (!p2.equals(p3) && !p2.equals(p4))) {
                                Line2D line2 = new Line2D.Double(p3.getX(), p3.getY(), p4.getX(), p4.getY());
                                if(line2.intersectsLine(line1)) return true;
                        }

                }
                return false;
        }



        public static Polygon2D GetRouteToHull(Point2D p) {
                int ind = 0;
                Polygon2D route = new Polygon2D();
                route.addPoint(p);
                List<Point2D> tmp = new ArrayList<Point2D>(Arrays.asList(obstacle.asPointsArray()));
                List<Point2D> intersections = new ArrayList<Point2D>();

                while(!IsOnHull(p)){
                        Double shortestdistance = Double.POSITIVE_INFINITY;
                for(int i=0; i<tmp.size(); i++) {
                                if(p.Distance(tmp.get(i))<shortestdistance) {
                                        shortestdistance=p.Distance(tmp.get(i));
                                        ind = i;
```

```
                        }
                }
        if(!Intersects(p, tmp.get(ind))) {
                route.addPoint(tmp.get(ind));
                p = tmp.get(ind);
                tmp.remove(ind);
                tmp.addAll(intersections);

        }
        else {
                intersections.add(tmp.get(ind));
                tmp.remove(ind);
        }

        };

        return route;
}


public static Polygon2D Route() {
        Polygon2D start = new Polygon2D(GetRouteToHull(points.sourcePoint()));
        Polygon2D end = new Polygon2D(GetRouteToHull(points.destinationPoint()));
        if(v == true) {
                drawRoute(start, Color.ORANGE);
                drawRoute(end, Color.ORANGE);
                resetcanvas();
        }
        Polygon2D route1 = new Polygon2D(start);
        Polygon2D route2 = new Polygon2D(start);
        int i=0;
        while(!hull.getIndex(i).equals(start.getIndex(start.size()-1))) i++; //get to hull point
        i++;
        while(!hull.getIndex(i).equals(end.getIndex(end.size()-1))){
                route1.addPoint(hull.getIndex(i));
                i++;
                if(i==hull.size()) i=0;
        }
        for(i=end.size()-1; i>=0; i--) route1.addPoint(end.getIndex(i)); //add path from hull to dest

        i=hull.size()-1;
        while(!hull.getIndex(i).equals(start.getIndex(start.size()-1))) i--; //get to hull point
        while(!hull.getIndex(i).equals(end.getIndex(end.size()-1))){
                route2.addPoint(hull.getIndex(i));
                i--;
                if(i==0) i=hull.size()-1;
        }
        for(i=end.size()-1; i>=0; i--) route2.addPoint(end.getIndex(i)); //add path from hull to dest

        if(v==true) {
                drawRoute(route1, Color.GREEN);
                resetcanvas();
                drawRoute(route2, Color.GREEN);
                resetcanvas();
        }

                route1 = optimizeRoute(route1);
                route2 = optimizeRoute(route2);

        if(v==true) {
                drawRoute(route1, Color.GREEN);
                resetcanvas();
                drawRoute(route2, Color.GREEN);
                resetcanvas();
        }



        if(getPolyDistance(route2)<getPolyDistance(route1)) return route2;
        return route1;
}


public static double getPolyDistance(Polygon2D poly) {
        double distance = 0;
        for(int i=0; i<poly.size()-1; i++) {
                distance+=poly.getIndex(i).Distance(poly.getIndex(i+1));
        }
        return distance;
}


public static void drawRoute(Polygon2D p, Color c) {
        StdDraw.setPenColor(c);
        Point2D ary[] = p.asPointsArray();
```

```
		for (int j=0; j<ary.length; j++) {
			if (j < ary.length-1)
				StdDraw.line(ary[j].getX(), ary[j].getY(), ary[j+1].getX(), ary[j+1].getY());
			if(v==true) {
				StdDraw.pause(delay);
			}
		}
}


public static Polygon2D optimizeRoute(Polygon2D poly) {
		Stack<Point2D> finalroutestart = new Stack<Point2D>();
		Stack<Point2D> finalrouteend = new Stack<Point2D>();

		finalroutestart.push(poly.getIndex(0));				//sets to source point
		finalrouteend.push(poly.getIndex(poly.size()-1));		//sets to dest point
		int i=1;

		while(!IsOnHull(finalroutestart.peek())) {
			if(Intersects(finalroutestart.peek(), poly.getIndex(i))) {	//if it intersects
				finalroutestart.push(poly.getIndex(i-1));	//add previous point
			}
			i++;
		}
		int k=poly.size()-2;
			//reverse for destination
		while(!IsOnHull(finalrouteend.peek())) {
			if(Intersects(finalrouteend.peek(), poly.getIndex(k))) {
				finalrouteend.push(poly.getIndex(k+1));

			}
			k--;
		}

		Point2D ary[] = new Point2D[finalroutestart.size()];
		for(int j=ary.length-1; j>=0; j--) {
			ary[j] = finalroutestart.pop();			//converts stack to polygon
		}
		Polygon2D ply = new Polygon2D(ary);
		i--;
		while(!ply.getIndex(ply.size()-1).equals(finalrouteend.peek())) {
			ply.addPoint(poly.getIndex(i));		//adds hull until it reaches destination path
			i++;
		}
		finalrouteend.pop();
		while(!finalrouteend.empty()) {			//empties destination path into polygon
			ply.addPoint(finalrouteend.pop());
		}
		return ply;
}


public static void resetcanvas() {
		StdDraw.clear();
		StdDraw.setPenColor(Color.BLACK);
		StdDraw.setPenRadius(0.008);
		StdDraw.point(points.sourcePoint().getX(), points.sourcePoint().getY());
		StdDraw.point(points.destinationPoint().getX(), points.destinationPoint().getY());
		obstacle.drawFilled();
		StdDraw.setPenColor(Color.RED);
		hull.draw();
		StdDraw.setPenRadius(0.006);
}




public static void main(String[] args) {


		points = new ObstacleMap(args[0]);

		if(args.length == 2) {
			if(args[1].equals("-v")) {
				v = true;
				obstacle = points.shape();
				hull = JarvisMarch.findConvexHull(points.shape());
				resetcanvas();
				Polygon2D route = Route();
				drawRoute(route, Color.MAGENTA);
				double pathLength = getPolyDistance(route);
				System.out.println("Length of path found = " + pathLength);
				return;
			}
```

```
                }

                long startTime = 0;
                obstacle = points.shape();
                obstacle.drawFilled();
                startTime = System.currentTimeMillis();

                hull = JarvisMarch.findConvexHull(points.shape());
                Polygon2D route = Route();
                drawRoute(route, Color.MAGENTA);

                double pathLength = getPolyDistance(route);
                final long runTime = System.currentTimeMillis() - startTime;
                System.out.println("Execution time (ms) = " + runTime);
                System.out.println("Length of path found = " + pathLength);
        }
}
```