EE324 Project 1

Convex Hulls and Path Finding

# Introduction

The purpose of this project is to create an algorithm which finds the convex hull of a given set of points, and implements a path finding algorithm. It must find the shortest distance from the given source point, along the hull to the given destination point. The provided code defines:

* A “Point2D” object, with x and y coordinates.
* A “Polygon2D” object containing a Point2D object array defined as vertices. It contains several useful methods used in future parts of the project:
  + addPoint() allows for a new Point2D object to be added to the Polygon2D object.
  + size() returns the number of vertices in the Polygon2D object.
  + getIndex() returns a Point2D object at an input index from the Polygon2D object.
  + asPointsArray() returns the Polygon2D vertices as a Point2D array.
  + draw() draws the Polygon2D object, connecting each vertex in the array to the next.
  + drawFilled() is the same as the draw() method, although it fills the Polygon2D object.
  + drawVertices() draws each of the vertices in the Polygon2D object.
  + toString() returns the vertices in the Polygon2D object as a String.
* The “MapFileReader” class reads each line of a given text file, taking the source point, destination point, and polygon points from the file separately.
* The “ObjectMap” class contains the source point, destination point and Polygon2D shape object in private variables. It also contains a draw() method which draws a Polygon2D of the input ObstacleMap object, the source point, and the destination.

# Jarvis March

The “JarvisMarch” class is used to implement the Jarvis march or “Gift Wrapping” algorithm. This algorithm finds the convex hull of the input ObstacleMap object. The Jarvis March algorithm works by finding an initial point on the convex hull – usually the point with the least or greatest x or y coordinate – and comparing each other point to this point, finding the most counter clockwise, and repeating the process from this point. This will give all the points along the convex hull. In order to implement this algorithm, a number of methods are required.

The findConvexHull() method is the method which implements all others. It takes the input Polygon2D object and converts it to an array of Point2D objects. It calls the getInitialPoint() method, then cycles through the Point2D array using a “for” loop within a “while” loop, finding the points along the convex hull by calling the isCounterClockwise() method. It exits the while loop when it has circumvented the hull, and has returned to the initial point, returning the hull points.

The initial point must first be found. The getInitialPoint() method uses an “if” statement within a “for” loop to cycle through each point of the polygon, finding that with the lowest x value, updating if it finds one with a lower value. If a point has an equal x value to the current lowest value, the y value is used to settle the tie, with the greatest y valued point taken as the initial point.

A ‘while(true)’ loop cycles through each point, using indexing integer values to pass three points into the isCounterClockwise() method.

The isCounterClockwise() method takes three points – the hull point, the next candidate hull point, and the point in the “for” loop –, comparing them to see which is more counter clockwise. If the tested point is more counter clockwise than the current candidate point, the Boolean value of true is returned.

If a value of true is returned, the ‘nextIndex’ variable is updated with the ‘for’ loops iterator value. If it is false, but the ‘collinear’ value is set to 1, the collinearDistance() method is called. This method takes in the point at the ‘currentIndex’, the point at the ‘nextIndex’, and the point at the ‘for’ loops current iterator value, ‘i’, and returns true if the point at ‘i’ is further than the point at ‘currentIndex’ than that at ‘nextIndex’. If a value of true is returned, the ‘nextIndex’ variable is updated with the ‘for’ loops ‘i’ value. After the loop is complete, the ‘currentIndex’ variable is updated with the ‘nextIndex’ value. If the break condition is not met, the point at the ‘currentIndex’ is added to the hull array.

The above methods give the basic functionality for the Jarvis March algorithm, and returns a Point2D array of hull points to the main funciton. Utilising the aforementioned drawMap() and draw() method within main() the desired ObstacleMap object can have a convex hull generated.

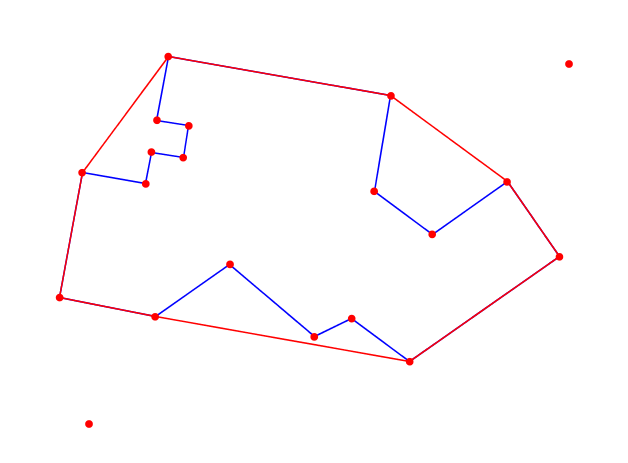


Figure 1: Generated Convex Hull for TestMap1A, with included Source and Destination Points

# Shortest Path

The implementation of a shortest path algorithm involves moving from the given source point, through the convex hull, to the destination point. In order to find the distance between two selected Point2D objects, a distanceTo() method is implemented. This method takes the distance from the two input Point2D objects using the distance formula . This distance value of type double is then returned.

The findShortestPath() method in the JarvisMarch class uses the createGraph() method to create a weighted visibility graph, giving the distance from each point to each other visible point in the form of an array of type double[][].

It firstly calls the hullGraph() method. This method iterates through each point in the passed in hull points object, testing for visible points. This is performed by passing the two hull points to the isIntersecting() method, along with the polygon points. If there is no intersection between the hull points and the polygon points, the value at the position [i][j] in the hull graph is set to the distance between the points at index ‘i’ and ‘j’. If there is an intersection, or ‘i’ is equal to ‘j’, the value at [i][j] is set to zero. The completed graph is then returned to the createHullGraph() method.

The isIntersecting() method takes in two Point2D objects, and the Point2D array of points. It then creates a Line2D object from the two Point2D objects, and using two ‘for’ loops, creates another Line2D object consisting of all combinations of points in the passed in array. The two lines are then tested for intersections using the intersectsLine() java built in function. If there is an intersection, the method returns true, and otherwise returns false. A number of exceptions must be checked due to the intersectsLine() function returning true if the same point is contained in both lines. This will cause all cases of hull points to return true. Therefore two ‘if’ statements check if there are duplicate points in the two Line2D objects.

Within the createGraph() method, a new graph of type double[][] is created, and the source points visibility is checked against all hull points, checking against the polygon points for cases of intersections. Similarly to the hullGraph() if the chosen hull point is determined to be visible, the distance value is added to the visibility graph. All the hullGraph values are then added to the new graph. The destination point is then checked in similar fashion to the source point. This graph is then returned to the findShortestPath() method.

Within the findShortestPath() method, a distance[] array is defined, with length equal to the graph length, and all values set to zero. A ‘nextIndex’ integer value is initialised as zero. A Polygon2D object, ‘path’, is created to contain the points of the shortest path. A Point2D[] array is created to hold the hull points, along with the source and destination points. An ArrayList of Point2D objects holds the points which have not yet been visited by the method while searching for the shortest path. A double value for the ‘pathDistance’ is set to ‘POSITIVE\_INFINITY’, to be used in the search process. Within the search each point is checked for visibility to each other point, and the distance compared to the ‘pathDistance’. If there is a visible point, with non-zero value, less than the ‘pathDistance’, the distance array value for the current ‘for’ loop iterator ‘i’ is set to the value in the graph at the ‘i’ and ‘j’ iterators. The ‘nextIndex’ value is then set to the value of ‘j’ as this is the current closest value to ‘i’. Once the closest point to ‘i’ is found, it is removed from the ‘unvisited’ ArrayList, and added to the ‘path’ Polygon. ‘i’ is then set to the ‘nextIndex’ value minus one, as one will be added back in when the ‘for’ loop repeats. The path is created by searching for each nearest point, with a break condition set for when the point at ‘nextIndex’ is equal to the destination point. The path is then drawn using a modified draw() method, called drawPath(), which draws the ‘path’ Polygon2D object minus the edge between the destination and source points.

The findShortestPath() method implements a NearestNeighbour style of path finding algorithm. It finds the nearest point to the current point, which does not increase the overall path distance, and sets this point as the next point to be searched from. This process continues until the destination point is reached.

# Case 1 – Collinear Test Map

Case 1 is the collinear test map. This map tests the convex hull method for how it deals with maps with collinear points. The test map is a square shape, with the source and destination points outside of the polygon. There are a number of points in the centres of the edges of the polygon, which act as collinear points in the findConvexHull() method.

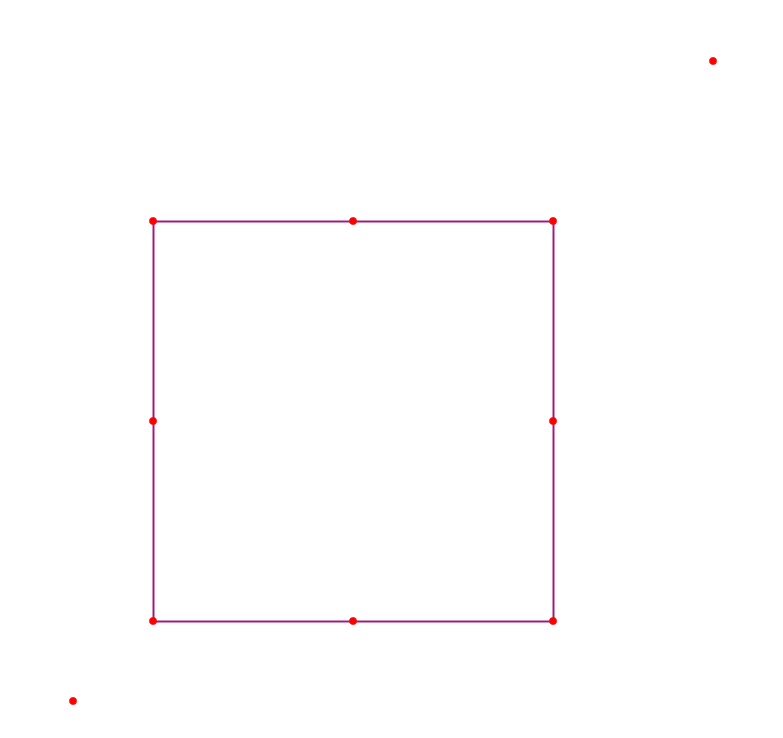


Figure 2: Collinear Test Map

For this case, the upper left point will be taken as the initial point, as it has both the smallest x value, and highest y value. From there, the top centre point will be the most counter clockwise. When the top right point is checked, it will return a false value, but the value of the variable ‘collinear’ will be set to one. Within the findConvexHull() method, this variable will be checked, and due to the value of one, will call the collinearDistance() method. This method will then return a value of true, as the top right point is a further distance from the top left point than the top centre point. This top right point is then added to the hull.

This process repeats along the square, as each side has a centre point which is collinear to the corner point. The findConvexHull() method will return the ‘hull’ Polygon2D object, which will consist of the four corner points of the square.

The findShortestPath() method will call the createGraph() method, which will in turn call the hullGraph() method. The returned hull graph from this method will show visibility from each corner to the adjacent corners. Using the exceptions in the ‘if’ statements, it will be seen that opposite corners cannot be visible from each other, while each corner can see its adjacent neighbour. This hullGraph is then returned to the createGraph() method.

The createGraph() method first tests the visibility of the source point. This point can see the bottom left, top left, and bottom right points, without any intersections occurring. The distance from the source points to each of these points is added to the new graph, with all other values set to zero. The hull graph is added to the new graph. The process for the source point repeats for the destination point, with the top left, top right, and bottom right points being visible from the destination without intersection. These values are added to the graph, with all other values set to zero, and the completed graph is returned to the findShortestPath() method.

The findShortestPath() method starts at the source point, and moves to the nearest neighbour, as found by searching for the smallest value from the source point in the graph. The next points nearest neighbour is then searched for in the same way. This repeats until the destination point is reached, satisfying the break condition and exiting the loop. The path is then drawn using the drawPath() method, and the path is returned.

The results from both the findConvexHull() method, and the findShortestPath() method can be seen in Appendix, Collinear Test Map, Table 1. Images of the hull, the visibility graph, and the path taken can be seen in Appendix, Collinear Test Map, Figure 2, Figure 3, and Figure 4.

# Case 2 – Source and Destination Outside Hull

Case 2 uses test map 1A. This test map contains a complex polygon, with a source and destination point outside of the convex hull. There are no collinear points in this test map, and as such the collinearDistance() method is not used.

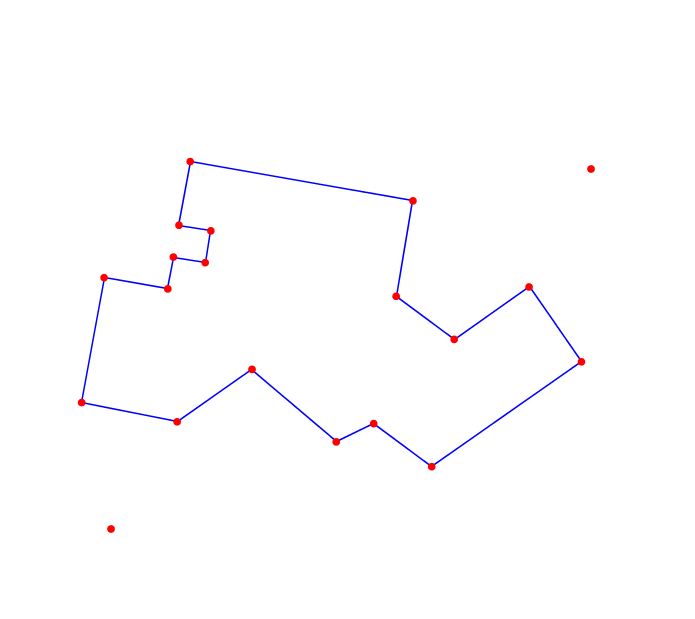


Figure 3: Test Map 1A

For this case, the bottom left point will be the initial point, as it is the polygon point with the smallest x value. The findConvexHull() method moves from this point to the most counter clockwise point. The hull found can be seen in Appendix, Test Map 1A, Figure 7.

The findShortestPath() method will call the createGraph() method, which will in turn call the hullGraph() method.

# Appendix

## Collinear Test Map

Table 1: Results of the Collinear Test Map – Hull Points, Visibility Graph Values, Path Taken, and Path Distance.

|  |  |  |  |
| --- | --- | --- | --- |
| Hull Points | Visibility Graph | Path Taken | Path Distance |
| |  | | --- | | (0.2,0.7) | | (0.7,0.7) | | (0.7,0.2) | | (0.2,0.2) | | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | i\j | 0 | 1 | 2 | 3 | 4 | 5 | | 0 | 0 | 0.608276253 | 0 | 0.608276253 | 0.141421356 | 0 | | 1 | 0.608276253 | 0 | 0.5 | 0 | 0.5 | 0.728010989 | | 2 | 0 | 0.5 | 0 | 0.5 | 0 | 0.282842712 | | 3 | 0.608276253 | 0 | 0.5 | 0 | 0.5 | 0.728010989 | | 4 | 0.141421356 | 0.5 | 0 | 0.5 | 0 | 0 | | 5 | 0 | 0.728010989 | 0.282842712 | 0.728010989 | 0 | 0 | | |  | | --- | | 0 | | 4 | | 3 | | 5 | | 1.369432345 |

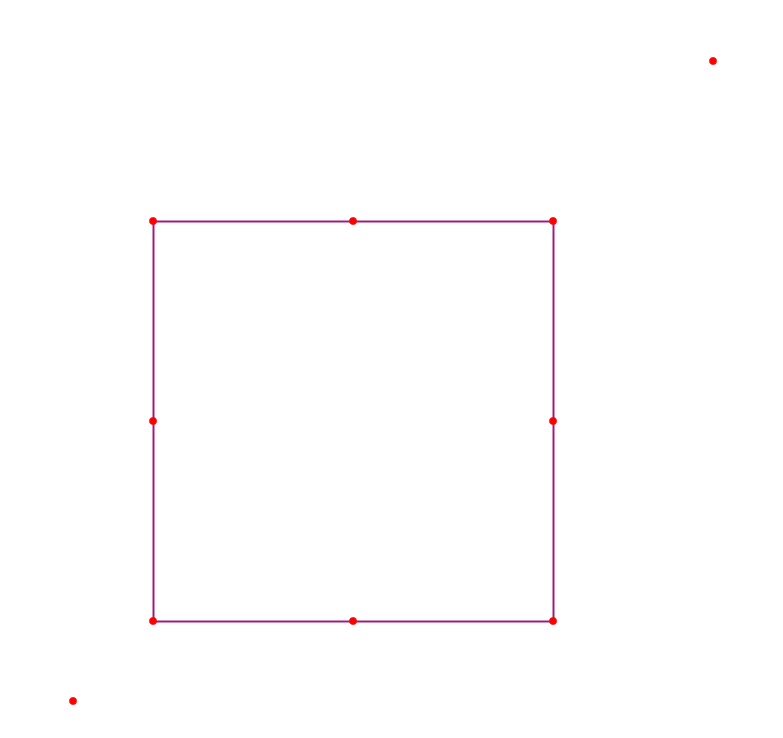


Figure 4: Collinear Test Map Convex Hull

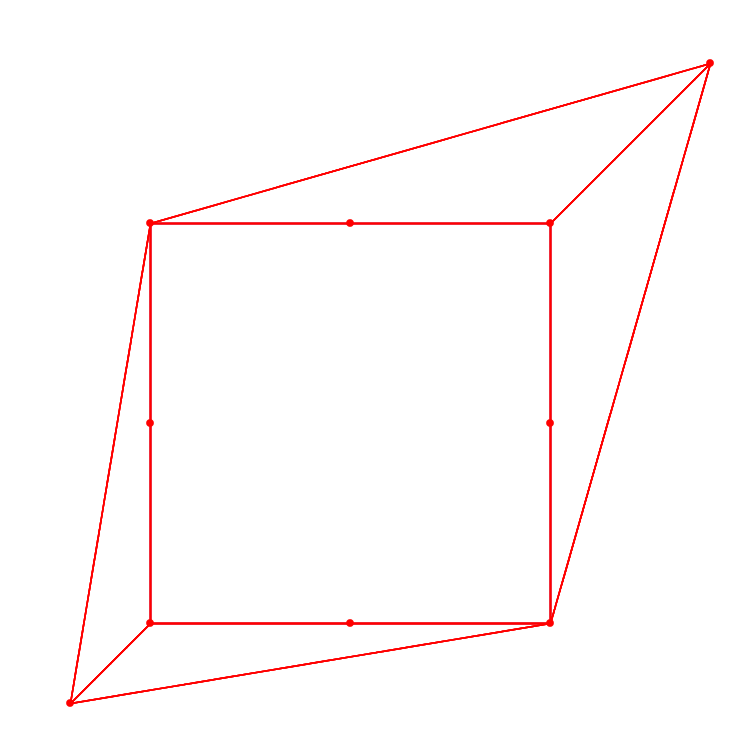


Figure 5: Collinear Test Map Visibility Graph

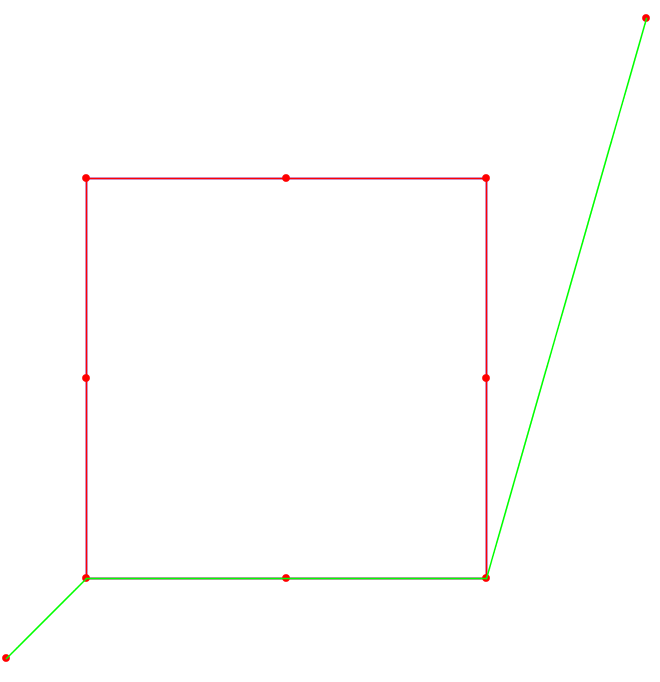


Figure 6: Collinear Test Map Shortest Path

## Test Map 1A

Table 2: Table 1: Results of the Test Map 1A – Hull Points, Visibility Graph Values, Path Taken, and Path Distance.

|  |  |  |  |
| --- | --- | --- | --- |
| Hull Points | Visibility Graph | Path Taken | Path Distance |
| |  | | --- | | (0.1633082713502252,  0.3580440919202139) | | (0.19136635565534096,  0.51416936087886) | | (0.29898704616327965,  0.6592655594084481) | | (0.5774562163471602,  0.6101639260127456) | | (0.7225524714297487,  0.502543235504807) | | (0.7880860072035134,  0.4089515831099261) | | (0.6009026862557517,  0.27788439037739576) | | (0.2826522291284597,  0.3340005347506271) | | |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | i\j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | 0 | 0 | 0.162247397 | 0 | 0 | 0 | 0 | 0 | 0.408398019 | 0.157440574 | 0 | | 1 | 0.162247397 | 0 | 0.158626466 | 0 | 0 | 0 | 0 | 0 | 0.121741829 | 0 | | 2 | 0 | 0.158626466 | 0 | 0.18065193 | 0 | 0 | 0 | 0 | 0 | 0 | | 3 | 0 | 0 | 0.18065193 | 0 | 0.282765007 | 0 | 0 | 0 | 0 | 0.501098624 | | 4 | 0 | 0 | 0 | 0.282765007 | 0 | 0.180651976 | 0 | 0 | 0 | 0.226081066 | | 5 | 0 | 0 | 0 | 0 | 0.180651976 | 0 | 0.114254285 | 0 | 0 | 0.166558149 | | 6 | 0 | 0 | 0 | 0 | 0 | 0.114254285 | 0 | 0.228508653 | 0 | 0.241342666 | | 7 | 0.408398019 | 0 | 0 | 0 | 0 | 0 | 0.228508653 | 0 | 0.323159984 | 0 | | 8 | 0.157440574 | 0.121741829 | 0 | 0 | 0 | 0 | 0 | 0.323159984 | 0 | 0 | | 9 | 0 | 0 | 0 | 0.501098624 | 0.226081066 | 0.166558149 | 0.241342666 | 0 | 0 | 0 | | |  | | --- | | 0 | | 8 | | 7 | | 6 | | 9 | | 0.950451877 |

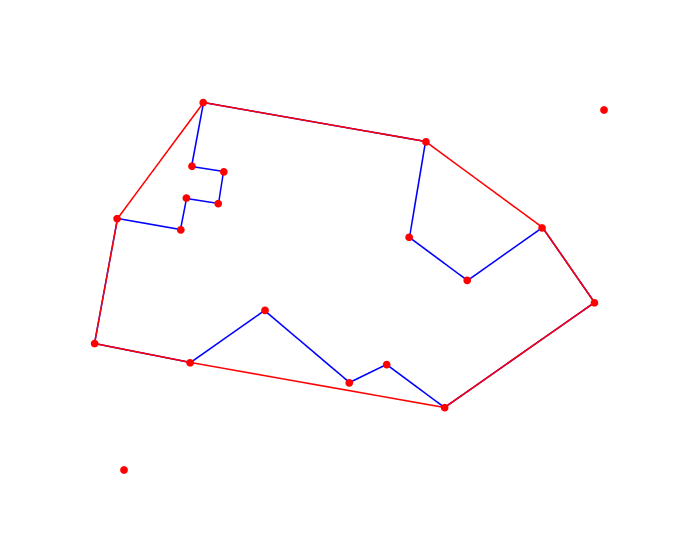


Figure 7: Test Map 1A Convex Hull

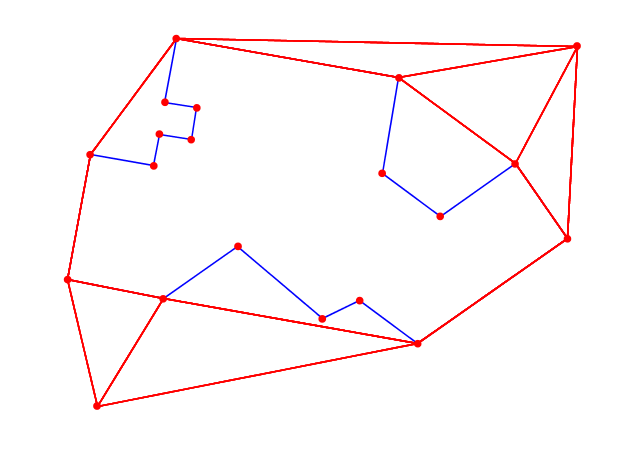


Figure 8: Test Map 1A Visibility Graph

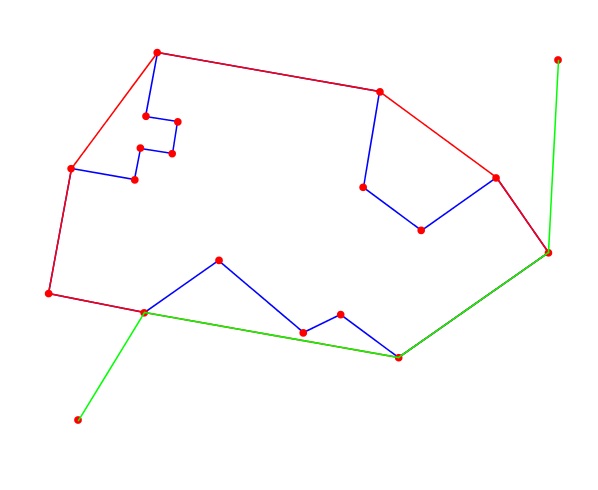


Figure 9: Test Map 1A Shortest Path