Genetic

Algorithm

Report

An Algorithm to Reduce

Edge-Crossings in Graphs

Group:

Adam Aherne 12159603

Alan Finnin 17239621

Daniel Dalton 17219477

William Cummins 17234956

# Introduction

# Fitness Functions

Our program offers the user a choice of two fitness functions. The default option is an implementation of the function outlined in the project specification. The second option is an implementation of the AngGA fitness measure (Toosi, et al., 2016).

## Default Function

The default fitness function calculates the total combined lengths of all edges for a given node ordering. The lower the value, the higher the fitness. Therefore, we are aiming to minimise this value.

To calculate the fitness cost of a layout, we must add the lengths of every edge produced by the individual. To do this we iterate through the individual, using nested loops, to find every possible pair of nodes. Using the node numbers as indices, we check the relevant location in the adjacency matrix to see if an edge joins these two nodes. If the nodes are connected, we calculate the distance of the edge and add it to the total, which is returned at the end of the function. This total starts as zero(0.0) and is a double precision floating-point value.

Before calculating the distance between the nodes, we must first calculate their cartesian coordinates. We do this in the same way as the *GraphVisualization* class provided in the specification*.* These values are then used with the basic formula for the length of a line:

When the function has finished iterating through the individual, and has calculated the length of all the edges, this value is returned as the fitness cost for the graph layout of that individual.

## AngGA Function

The user may also choose to use the AngGA based fitness function. This is a more complex function that involves calculating all edge lengths, all of the inside angles between edges, and the betweenness centrality of each node. The lower the value returned by the AngGA algorithm, the higher the fitness. The fitness cost for an ordering of nodes, *O*, is given by the following equation:

This equation is for a graph, , with set of vertices  {}, and a set of edges . The angle between edges and is represented by , while and represent the betweenness centralities of and respectively. The lengths of edges and are represented by and .

The betweenness centrality for each node of the graph must be calculated before this fitness function can be used. If the user chooses to use this function, we calculate these values and store them in a static array. To do this, we created a function, *getBetweenessCentralities*, and a helper function, *populatePaths*. The betweenness centrality of a graph node, , is the fraction of all the shortest paths between other nodes that pass through .

In these functions, we first iterate through the upper half of the adjacency matrix, examining all possible pairs of nodes. We can ignore the bottom half as the matrix is symmetric. If two nodes are directly connected, we continue the loop, as the path between them will consist of only one edge and no other nodes. If the nodes are not directly connected, we use or helper function. Before we call the helper function, we store all the possible branches from the starting node. We add all of the nodes in these branches in an array representing a combined open and closed list. This is then passed, along with all our starting branches, to the helper function. This function uses depth-first search to find paths between the current pair of nodes. This is why the open and closed lists are necessary.

In the helper function, we use depth-first search to follow all of the initial branches, which have been passed to us by the driver function. We follow each path until the target node is reached. We make sure all paths have been followed to same depth, to be certain no shortest path has been missed. It is also important to add a new path when branch points are met. Those paths that do not reach the target node are removed from the list. All paths that are not in the set of shortest paths are also removed. We calculate how many times each node appears in these paths. We then divide these values by the number of shortest paths we have found. These final values are added to the relevant locations in our static betweenness centrality array. The helper function then clears the open and closed list and returns control to the driver function, which continues its iteration through all pairs of nodes.

Once all the betweenness centralities have been worked out, they can be used in the fitness cost calculation. The fitness function iterates through each inside angle formed by edges in the graph. It calculates the size of the angle and the lengths of the edges which form it. These values are then filled into the AngGA equation, given above. The value given by the AngGA equation is added to the return value of the fitness function. This value starts as zero(0.0) and is a double precision floating-point number

# Evaluation

# Conclusion