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

Many problems associated with graphs and their layout are NP-hard. This includes the problem of minimizing edge-crossings (Tettamanzi, 1998). Minimizing edge-crossings is just one criterion for the layout of a graph. Generally, these criteria aim to improve the aesthetics of a graph, readability being a prime measure of this. This project focuses only on the reduction of edge-crossings. If one were to compare a randomly selected layout of the input graph for this project against a layout produced by our genetic algorithms, the difference in readability would be extremely clear.

Genetic algorithms are commonly used to generate solutions to optimization and search problems (Mitchell, 1998). In this project we are looking to optimize the number of edge-crossings produced from the set of possible node orderings of our input graph. There is a large body of work related to genetic algorithms for graphs available. The most important part of a genetic algorithm is its fitness function. This provides a heuristic measurement of the optimization of a node ordering. Heuristics are the most suitable option for solving NP-hard optimization problems, such as minimizing edge-crossings (Apter, 1970). This is why genetic algorithms are so important when working with graphs.

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 segment:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

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:

$$f(O) = \sum_{v_i \in V} \sum_{\substack{v_j, v_k \in V \\ \{v_i, v_j\} \in E \\ \{v_i, v_k\} \in E}} a_{ijk} b_i b_j b_k e_{ij} e_{ik}$$

This equation is for a graph, $G = (V, E)$, with set of vertices $V = \{v_1, v_2, \dots, v_n\}$, and a set of edges $E \subseteq V \times V$. The angle between edges $\{v_i, v_j\}$ and $\{v_i, v_k\}$ is represented by a_{ijk}, while b_i, b_j and b_k represent the betweenness centralities of v_i, v_j and v_k respectively. The lengths of edges $\{v_i, v_j\}$ and $\{v_i, v_k\}$ are represented by e_{ij} and e_{ik}.

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 driver function, *getBetweenessCentralities*, and a helper function, *populatePaths*. The betweenness centrality of a graph node, v, is the number of shortest paths in the graph that pass through v.

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 call our 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

To evaluate our two fitness functions, we compared them in respect to two aspects. We first designed tests to compare how effective and efficient they were in reducing edge crossings in the given graph. Then we analysed the program code and compared the function runtimes asymptotically.

Effectiveness & Efficiency

Our group created a set of tests designed to analyse the effects of each of the input parameters on the effectiveness of the program. To achieve this, we first decided on base values for each of the inputs. Each test would alter a chosen parameter, while the other three would remain at their base values. The program would be run ten thousand(10,000) times with these input values; five thousand(5,000) times using the default fitness function, and five thousand(5,000) times using our AngGA implementation. To accomplish this, we created a program that automated our testing. Table 1 shows the results using the base values. This includes the average number of edge-crossings produced by the fittest individual of the final generation for each function, as well as the lowest number of crossings and the highest number crossings. Table 2 shows the results for the rest of the tests, each of which aim to measure the effect of a particular input parameter. The base values we chose for the input parameters are:

- $P = 50$
- $G = 50$
- $Cr = 25$
- $Mu = 25$

After the tests were complete, both algorithms had produced a best result of one edge-crossing. We continued adjust the input values, with the aim of finding a solution. However, we could not find a complete solution. One was the lowest number of crossings our program could achieve. Table 3 shows the minimum number of generations required to achieve this result for both fitness functions, as well as the other input parameters used.

Input Values				Edge-Crossings: Default				Edge-Crossings: AngGA			
P	G	Cr	Mu	Mean	Median	Lowest	Highest	Mean	Median	Lowest	Highest
50	50	25	25	6.6538	6	1	19	13.0708	12	1	38

Table 1: Results from Base Values

Input Values				Edge-Crossings: Default				Edge-Crossings: AngGA			
P	G	Cr	Mu	Mean	Median	Lowest	Highest	Mean	Median	Lowest	Highest
50	50	25	25	6.6538	6	1	19	13.0708	12	1	38
100	50	25	25	5.098	5	1	15	10.381	10	1	36
50	100	25	25	4.183	4	1	13	9.5572	9	1	34
50	50	50	25	6.3522	6	1	18	12.553	12	1	43
50	50	25	50	6.5484	6	1	6	12.1572	11	1	50

Table 2: Tests to Measure Effect of Input Parameters

Input Values: Default				Input Values: AngGA			
P	G	Cr	Mu	P	G	Cr	Mu
200	18	45	45	225	24	55	45

Table 3: Fewest Generations to Reach One(1) Edge-Crossing

Asymptotic Comparison

While the input file given is quite small, it is important to consider how our two fitness functions would perform when working on large datasets. There is an order of magnitude in the difference between the two functions. The default function has running time in the order of $O(n^2)$, while AngGA is in the order of $O(n^3)$.

The default function uses nested for loops, and the body of this loop is constant time. The inner loop does not begin at zero, rather it starts at the outer loop index plus one. This means that on its first iteration, the inner loop will run $n-1$ times, then $n-2$ times, and so on. This means the loop runs a number of times equal to the sum of the first $n-1$ numbers. This can be visualized as follows:

$$\underbrace{(n-1)+(n-2)+\cdots\big(n-(n-2)\big)+(n-(n-1))}_{n-1}$$

Using Gauss' formula this evaluates to $\frac{n^2-n}{2}$. Removing the lower order terms leaves us with $O(n^2)$.

The AngGA function is similar to the default, as the body of the loop is constant time. However, the loops with runtime $\frac{n^2-n}{2}$ are inside another loop, which iterates n times. Multiplying these together gives us $\frac{n^3-n^2}{2}$. Removing the lower order terms leaves us with $O(n^3)$. It is also worth noting that to use the AngGA function we must first calculate the betweenness centralities. The function to do this is called only once, but also has runtime $O(n^3)$.

Conclusion

The default fitness function outperformed the AngGA implementation in each test and required fewer generations to reach a result with only one edge-crossing. Our results suggest that there is also a slight difference between the two functions, in regard to how they are affected by the input parameters. Increasing the mutation rate appears to have a slightly greater positive effect on the results for the default algorithm. However, the opposite is true for the AngGA implementation. These differences are very slight, less than 0.2 between the means for the default function, and less than 0.4 for AngGA. An increased number of generations had the greatest positive effect on both fitness functions, and produced the lowest average edge-crossings.

Two important factors must be noted. The first is that we chose somewhat small input parameters to test our fitness functions. This is because at higher values the average number of edge-crossings became so low that there were no significant, observable differences between the different tests. The second factor to be considered is the importance of the initial population, $G(0)$. The fitness of this generation was a far greater factor than any of the input parameters, particularly on small to medium values of G or P. Table 1 shows that a one edge-crossing result was found with the base input parameters for both functions. Table 3 shows that this result can be achieved with as few as 18 generations. As the generation of individuals in $G(0)$ is random, this can skew results. This is shown by the large differences between the best and worst cases in each test, and by the fact that some tests produced an individual with worse fitness than other tests, despite having better average fitness. This is why we include the mean and the median in our results, as the median is less affected by extreme values.

Bibliography

Apter, M. J., 1970. *The Computer Simulation Of Behavior.* 1 ed. London: Hutchinson & Co..

Mitchell, M., 1998. *An Introduction to Genetic Algorithms.* 1 ed. Cambridge, Massachusetts: MIT Press.

Tettamanzi, A., 1998. *Drawing Graphs With Evolutionary Algorithms.* s.l., Springer, pp. 325-337.

Toosi, F. G., Nikolov, N. S. & Eaton, M., 2016. *A GA-Inspired Approach to the Reduction of Edge Crossings in Force-Directed Layouts.* s.l., s.n., p. 89.