An Evolutionary Approach to the Graph Bandwidth Problem

Maaz Saeed
Dhanani School of Science and Engineering
Habib University
Karachi - 75290, Sindh, Pakistan
Email: @st.habib.edu.pk

Muhammad Usaid Rehman
Dhanani School of Science and Engineering
Habib University
Karachi - 75290, Sindh, Pakistan
Email: mr04302@st.habib.edu.pk

Maham Shoaib Patel
Dhanani School of Science and Engineering
Habib University
Karachi - 75290, Sindh, Pakistan
Email: mp04911@st.habib.edu.pk

***Abstract*—The bandwidth problem for a graph is an NP-complete problem. Almost identical to the bandwidth problem for matrices, it finds applications in sparse matrix handling, and electronic design automation. It is a combinatorial optimization problem which is also NP-hard to approximate. Therefore, we attempt to solve the bandwidth problem using a basic evolutionary algorithm and observe the results obtained.**

***Keywords*—**Graph Theory, Bandwidth, Graph Bandwidth, Evolutionary Algorithms

1. Introduction

The graph bandwidth is a well-studied problem in graph theory. It is a combinatorial optimization problem where the objective is to minimize the maximum distance between two vertices of a graph by finding a suitable labelling $f : V(G) \to \{1, 2, \ldots, n\}$. There are two different forms of the bandwidth problem – the bandwidth problem for graphs and the bandwidth problem for matrices. Both versions of the problem are closely related since the graph version of the problem can be reduced to the matrix version using the adjacency matrix of the graph. The problem can be visualized as placing the vertices of a graph at distinct integer points along the x-axis so that the length of the longest edge is minimized.

The inception of the matrix bandwidth problem occurred in the 1950s when structural engineers attempted to analyze steel frameworks by their structural matrices via computerized manipulation. The term 'bandwidth' was birthed as the engineers had endeavoured to discover a matrix in which all the non-zero elements lay withing a narrow 'band'. The inspiration for this came from operations such as inversion or finding determinant in as little time as possible.

In 1962, similar to this approach, L.H Harper, and A.W Hales conceived the bandwidth, and bandwidth sum. They used edge differences to represent single errors in a 6-bit picture code, in a hypercube, where it's vertices were words of the code [1]. Some time after this, R.R Kohrfage initialized his work on the graph bandwidth problem [2]. Finally, F. Harary published the problem, as we know it today, officially [3].

1.1. Formal Definition

In more formal terms, the mapping f is defined as $f : V(G) \to \{1, 2, \ldots, n\}$, where $n = |V|$. This is also called a *proper numbering* of the graph G. [4] Therefore, we can think of these mappings as essentially labelling or numbering of the vertices. The bandwidth of a numbering f is defined as:

$$B_f(G) = \max_{uv \in E(G)} \{|f(u) - f(v)|\} \tag{1}$$

The bandwidth of the graph G is given by the bandwidth of the best possible numbering:

$$B(G) = \min\{B_f(G) : f \text{ is a numbering of } G\} \tag{2}$$

Bandwidths can be computed using any integer mapping, however, to make our implementation easier, we will be restricting ourselves to working with proper numberings only.

There exist several known mathematical bounds that relate the bandwidth of a graph to various graph theoretic properties. For example, it is a simple exercise to prove that

$$B(G) = n - 1$$

where G is a complete graph of the form K_n. Similarly, there are also bounds relating the chromatic number and diameter with the bandwidth of the graph. [2]

1.2. NP-Completeness of the bandwidth problem

The bandwidth problem itself is a special case of the quadratic bottleneck assignment problem, which is known to be NP-hard. It is also known that the bandwidth problem is NP-hard to approximate which makes it impossible to find an $O(1)$-approximation algorithm for the bandwidth problem.

Historically, there have been endeavors to decrease (see [5], [6]), or minimize (see [7], [8]) the bandwidth of large, spares matrices, effectively, by permuting rows and columns. This has been translated to graph theory by Harary (see [9]). Papadimitriou proved that the minimization of the bandwidth of a matrix is an NP-complete problem. [10]

There are several heuristic algorithms for the bandwidth problems such as the Cuthill-McKee algorithm, and the Gibbs-Poole-Stockmeyer algorithm. Heuristic approaches are of interest to us since they will give us an idea as to how we can design appropriate meta-heuristic techniques to solve the bandwidth problem.

2. Evolutionary Algorithms

Evolutionary Algorithms is a term referring to a family of algorithms based on the evolution we see around us in nature. By mimicking learning, natural selection, and reproduction, we can produce solutions for various search and optimization problems. This concept of evolving algorithms enables us to bypass the setbacks of traditional search/optimization algorithms.

2.1. Darwinian Evolution

Evolutionary Algorithms are derived of a simplified Darwinian evolution. The principles of such a cycle can be as such:

1) **Variation** - Individual members of a given population may have differing attributes from one another, for example, physical appearance.
2) **Inheritance** - Offspring resemble their parents to certain extents. In this manner, traits are passed down from one generation to another; unrelated individuals are less likely to have common traits, as compared to them with their family trees.
3) **Selection** - Nature follows 'survival of the fittest' ideology. Individuals that are better able to locate and make use of resources compared to their peers, are more likely to survive in their respective environments.

In accordance with these principals, results that we obtain from our algorithm may or may not resemble those of previous iterations. With careful manipulation of parameters, solutions produced by our code can be ranked higher or lower than others.

2.2. Analogies

Where Darwinian evolution maintains a population of individual solutions, genetic algorithms maintain *individuals* - a population of candidate solutions [11]. The theory behind these algorithms is that solutions are produced, and improved upon, by iteratively re-producing newer generations of solutions.

The various components of an evolutionary algorithm are as follows:

1) **Genotype** - In nature, genotypes are collections of genes. When two individuals procreate, a mixture of genes from both, will make up the chromosomes of the offspring. In code, these **chromosomes** can be expressed, for example, as strings in binary.
2) **Population** - Population refers to the collection of chromosomes. At any given moment, the algorithm will maintain a population of individuals - candidate solutions for the problem that is being attempted to be solved. In a nutshell, it is the current generation, which will be replaced by the next generation of offspring.
3) **Fitness Function** - a function used to evaluate individuals in a given population. Individuals that produce better results, will be more favoured when it comes to selection for breeding of newer generations. As this cycle runs, individuals display continuous improvement until a satisfactory solution to our problem is found, at which point we can terminate the operation.
4) **Selection** - After individuals are evaluated and awarded a *fitness value*, the best among them are chosen to breed and produce the newer generation. It is important to note that individuals with lower scores are still selected, but with lower probabilities, so as to not cause extinction of their respective attributes.
5) **Crossover** - refers to the mixing of chromosomes of the two parent individuals that were paired in the selection process to produce two new chromosomes (offspring). This process is also known as recombination.
6) **Mutation** - fulfills the purpose of periodically (at random; not in a set pattern) refreshing the population. This introduction of new patterns in the chromosomes encourages the algorithm to search in unexplored areas, rather than just exploiting what it has already chartered. The mutation may occur as random changes in chromosomes, for example, in binary string

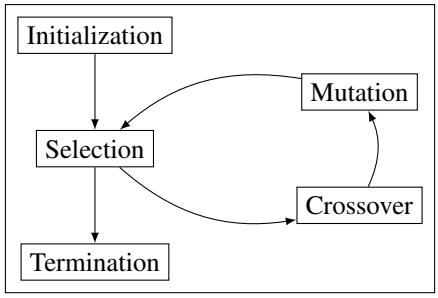


Figure 1: Flowchart of Evolutionary Algorithms

3. Implementation Details

3.1. Population Represenation

In our implementation, an individual chromosome in our population is stored in the form of key-value pairs[1], where

1. In Python, this is implemented as a dictionary

the key is the vertex and the value is the integer label of the vertex.

The population can be expressed as the set

$$\{(v_i, f(v_i)) : \ v_i \in V(G)\}$$

where v_i is a vertex in graph G and $f(v_i)$ is the integer label of the vertex. To initialize the population, we generated individuals using randomized numberings. Therefore, each individual is a proper numbering of the graph (see Section 1.1). While storing the numberings, we also compute the fitness of each individual (see Section 3.2) and store it along with each individual.

3.2. Fitness Function

The fitness function in our implementation was derived simply from the problem. Each individual was a proper numbering of the graph, and therefore, each individual had a corresponding bandwidth as described in Section 1.1. Therefore, we simply iterated over the edges of the graph and then found the maximum value. We can also say that we found the bandwidth $B_f(G)$ for a single numbering/ordering f.

Algorithm 1 Fitness Function

Input: A chromosome
Output: The fitness (bandwidth) of the chromosome
Initialize edge map:
$A \leftarrow \{(uv) : 0\} \quad \forall uv \in E(G)$
Compute fitness:
for each $uv \in E(G)$ **do**
  $A[uv] \leftarrow |f(u) - f(v)|$
end for
fitness $\leftarrow \max(A)$
return fitness

3.3. Selection

We implemented several different selection schema for both parent selection, and survivor selection. The schema we implemented were:

1) *Fitness-proportional Selection (FPS):* This selection scheme is also called roulette-wheel selection. Parent chromosomes are selected stochastically based on their fitness – parents with a higher fitness value have a greater change of being selected. Since we are doing a minimization problem, we scale the fitness values in a way that individuals with a lower fitness have a higher chance of being selected.
2) *Rank-based Selection (RBS):* In RBS, we sort the population by their fitness values and create a rank ordering. The selection probabilities are determined based on ranking and not on actual fitness values which can help reduce selective pressure.
3) *Binary Tournament Selection (BT):* BT is a specific case of the more general tournament selection, where a group of individuals are selected from the population and then the best individual among the sample is returned. In binary tournament, two individuals are selected from the population randomly, and the more fit individual of the two is returned. This reduces the selective pressure since it does not let the fittest individual dominate.
4) *Truncation/Elitism:* In truncation, the population is ordered according to fitness values. The ranking depends on the nature of the problem – minimization or maximation – and the individuals with the worst values are truncated to maintain a given population size.
5) *Random Selection:* In this selection scheme, parents are selected randomly based on a uniform random distribution where each individual has the selection probability of $\frac{1}{n_p}$ where n_p is the population size.

3.4. Crossover & Mutation

We performed a two-point crossover in order to bring genetic diversity to the population. All offspring was generated using the crossover operation, i.e. no offspring were copies of their parent chromosomes. A generic crossover method could not be used since the numbering had to be unique[2].

This method was implemented using various methods for lists and dictionaries that are available in Python. We present a pseudocode for our procedure in Algorithm 2.

To further maintain genetic diversity to encourage exploration of the solution space, we used a mutation operator. Based on a mutation probability rate that we keep around 0.2, we mutate our offspring chromosomes by swapping the label of two randomly chosen vertices.

Algorithm 2 Crossover

Input: Two parent chromosomes
Output: Two child chromosomes
childA1, childA2, childB1, childB2 $\leftarrow$ [ ]
numA, numB $\leftarrow$ Extract numberings from parent chromosomes
$a, b \leftarrow$ Randomly generate crossover points
$n \leftarrow |V(G)|$
for $i \leftarrow a$ to $i \leftarrow b$ **do**
  childA1.ADD(numA[i])
  childB1.ADD(numB[i])
end for
childA2 $\leftarrow$ leftover items from numA
childB2 $\leftarrow$ leftover items from numB
childA $\leftarrow$ childA2[:a] + childA1 + childA2[b:]
childB $\leftarrow$ childB2[:a] + childB1 + childB2[b:]
childA $\leftarrow$ CREATECHILD(childA)
childB $\leftarrow$ CREATECHILD(childB)

2. Similar to the Travelling Salesperson Problem.

4. Experimental Analysis

4.1. Testing Methods

4.1.1. Various Selection Schema. There are several parameters that can be controlled while implementing an evolutionary algorithm. To investigate our results and to determine which selection schema work best, we plotted different results by keeping some parameters constant. We kept the mutation rate, and the number of iterations constant. The number of generations are changing because we We varied the parent selection strategy and the survivor selection strategy. For results, see Section 4.2.

4.1.2. Parameter Selection. We kept some parameter values constant to test our implementation. The constant values were as follows:

1) `number_of_iterations` = 10
2) `population_size` = 100
3) `offspring_size` = 50
4) `mutation_rate` = 0.2

4.1.3. Graphic User Interface (GUI). To make the solution widely accessible, we developed an interface based on the `PyQT5` library. Our aim was to provide a user-friendly medium to run the algorithm and view results without having to go through extensive code.

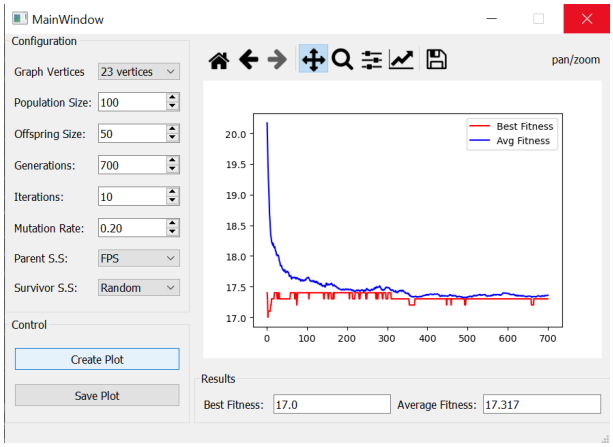
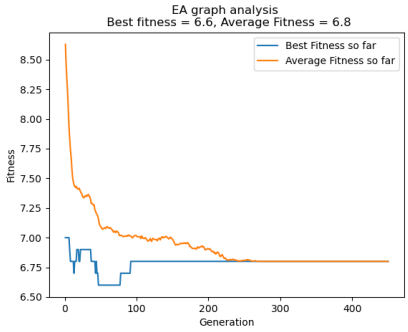


Figure 2: Snippet of the program's GUI

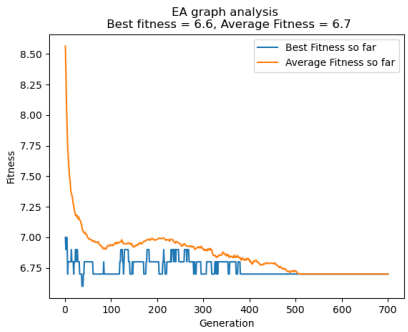
4.2. Results

We tested our algorithm on three different graphs, with 11, 23 and 100 vertices respectively. For the sake of brevity, we have only included results which followed an elitism (truncation) survival scheme. We plotted the best fitness and the average fitnesses of the algorithm against each generation averaged over 10 iterations. The number of generations for each plot varied slightly although it lied mostly in the 450-750 range due to varied convergence. We present the results here:

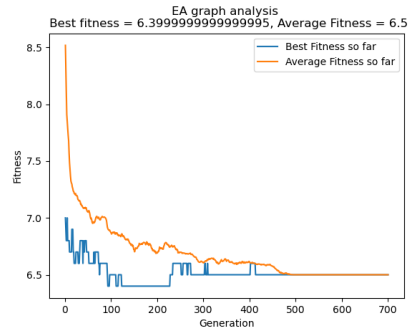
(i) Testing on 11-vertices graph:



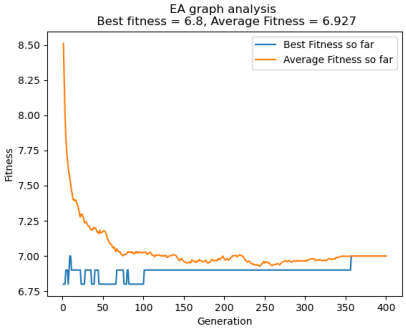
(a) Fitness-Proportional Selection



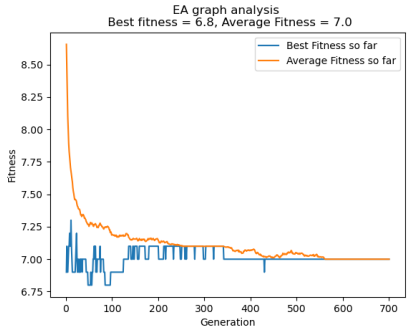
(b) Rank-Based Selection



(c) Binary Tournament Selection



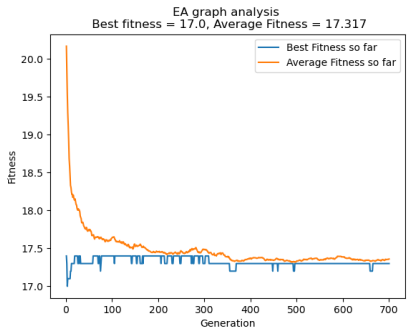
(d) Truncation/Elitism Selection



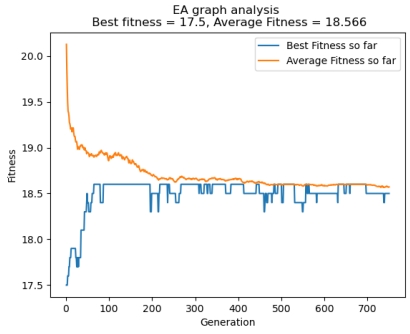
(e) Random Selection

Figure 3: Results of EA applied to 11-vertex graph

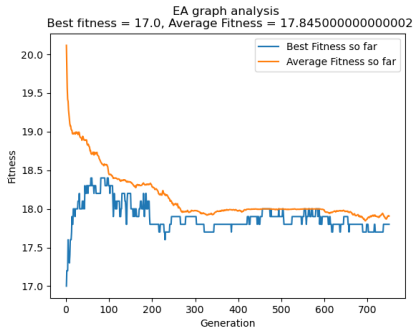
(iii) Testing on 23-vertices graph:



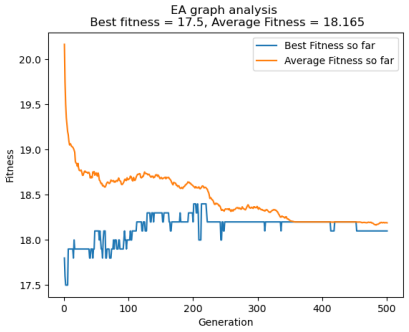
(a) Fitness-Proportional Selection



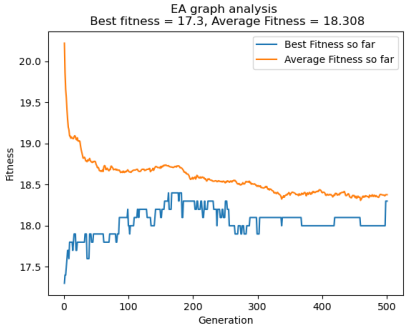
(b) Rank-Based Selection



(c) Binary Tournament Selection



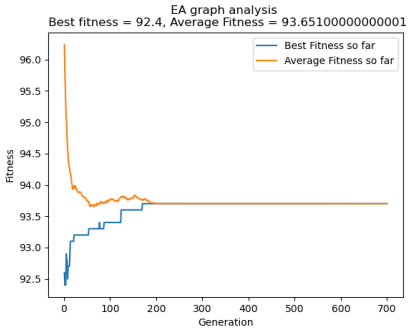
(d) Truncation/Elitism Selection



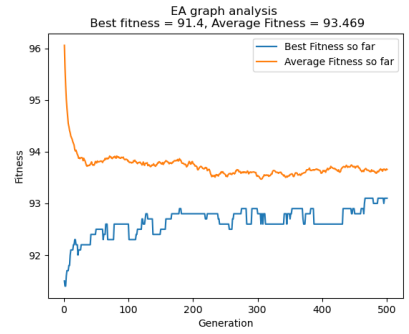
(e) Random Selection

Figure 4: Results of EA applied to 23-vertex graph

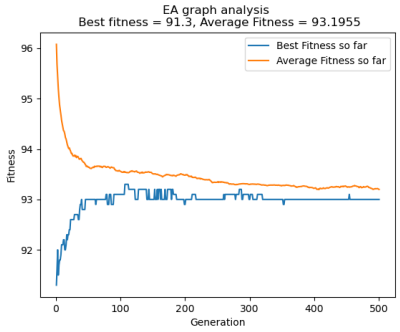
(ii) Testing on 100-vertices graph:



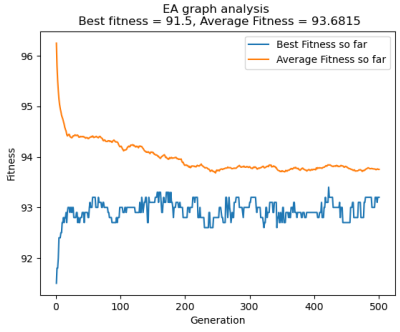
(a) Fitness-Proportional Selection



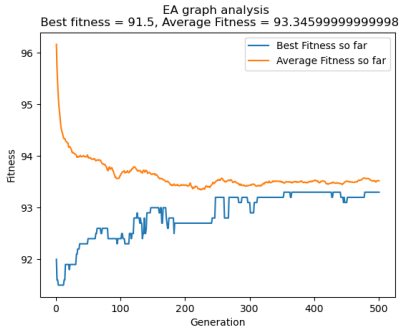
(b) Rank-Based Selection



(c) Binary Tournament Selection



Truncation/Elitism Selection



(d) Random Selection

Figure 5: Results of EA applied to 100-vertex graph

5. Conclusion

Judging by the results in Section 4.2, we can see that the graph bandwidth problem is not easy to solve. While we could obtain numberings for the graphs with 12 and 23 vertices, it was quite hard to do so with the graph with 100 vertices. This particular graph was also dense, which made the bandwidth quite high and difficult to minimize. On the other hand, the problem shows promising results if the graph itself is relatively sparse. For smaller graphs, the choice of parent selection did not have much impact on the results. However, for larger graphs, truncation seemed to perform better.

To further improve upon and to gain more understanding regarding evolutionary approaches to the bandwidth problem, we suggest comparing the results of evolutionary algorithm solutions to this problem with results obtained from heuristic algorithms described in Section 1.2, while also experimenting on different types of graphs and testing whether already known mathematical results regarding the bandwidth problem are being obeyed. Furthermore, we also plan on implementing different metaheuristics such as Ant Colony Optimization and more sophisticated forms of evolutionary computation in order to achieve better results.

The accompanying code for this project can be accessed at https://github.com/m-usaid/CS451-FinalProject.

Acknowledgments

We would like to acknowledge the freely available datasets that we obtained from two different online resources. The graphs were obtained from https://mat.gsia.cmu.edu/COLOR/instances/ and http://people.brunel.ac.uk/~mastjjb/jeb/orlib/colourinfo.html.

References

[1] L. H. Harper, "Optimal assignments of numbers to vertices," *Journal of the Society for Industrial and Applied Mathematics*, vol. 12, no. 1, pp. 131–135, 1964.

[2] P. Z. Chinn, J. Chvátalová, A. K. Dewdney, and N. E. Gibbs, "The bandwidth problem for graphs and matrices—a survey," *Journal of Graph Theory*, vol. 6, no. 3, p. 223–254, 1982.

[3] E. M. Fels, "Fiedler, m. (ed.): Theory of graphs and its applications, proceedings of the symposium held in smolenice in june 1963. publishing house of the czechoslovak academy of sciences, prag 1964. 234 s., preis kč 26,50," *Biometrische Zeitschrift*, vol. 8, no. 4, pp. 286–286, 1966.

[4] J. R. Lee, *Graph Bandwidth*, pp. 866–869. New York, NY: Springer New York, 2016.

[5] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*, ACM '69, (New York, NY, USA), p. 157–172, Association for Computing Machinery, 1969.

[6] T. R. P., "Sparse matrices," vol. 99, no. 3, pp. 66–74, 1973.

[7] K. Y. Cheng, "Minimizing the bandwidth of sparse symmetric matrices," *Computing*, vol. 11, no. 2, pp. 103–110, 1973.

[8] K. Y. Cheng, "Note on minimizing the bandwidth of sparse symmetric matrices," *Computing*, vol. 11, no. 2, pp. 27–30, 1973.

[9] "References," in *Sparse Matrices* (R. P. Tewarson, ed.), vol. 99 of *Mathematics in Science and Engineering*, pp. 141–151, Elsevier, 1973.

[10] C. H. Papadimitriou, "The np-completeness of the bandwidth minimization problem," *Computing*, vol. 16, no. 3, p. 263–270, 1976.

[11] W. E., *Hands-on Genetic Algorithms with Python*. UK: Packt Publishing Ltd, 1 ed., 2020.