A Report on

Using Genetic Algorithms for Vertex Colouring Problem

Written by Neel Dhapare
2020A7PS1223G

Contents

Introduction

This report discusses the use of the genetic algorithm, and subsequently, its variation, in solving the vertex colouring problem for 3 colours. For a given graph, the vertex colouring problem aims to maximise the number of vertices that are coloured such that no two adjacent vertices are coloured the same.

In accordance to the requirements of the assignment, the report shall limit itself to two versions of the genetic algorithm: one being the base template of a genetic algorithm as given in *Artificial Intelligence – A Modern Approach (4th ed.)*, and the other an improvised version that builds upon certain ideas that delve into the specificities of the given problem. For implementation, Python programming language will be used.

Base Genetic Algorithm

As mentioned above, the first variation of genetic algorithm that we shall use is the base template provided in *Artificial Intelligence – A Modern Approach (4th ed.)*. At its core, a genetic algorithm consists of four steps, some of which are repeated over and over until the required result is achieved, or until the result cannot be improved anymore. They are:

1. Population production
2. Selection
3. Recombination
4. Mutation

For optimisations, we use the term **fitness score** as a measure of how "good" a solution state is: a better fitness score corresponds to a superior solution state. In our case, the fitness score of the vertex colouring problem will be defined as the number of vertices that are coloured such that no two adjacent vertices are coloured the same.

The genetic algorithm attributes its effectiveness to the assumption that any given solution state has "blocks" that perform certain useful functions, which in turn improve the result. It attempts to ensure that these "blocks" survive over mutiple generations and pass on their functionality to the future solution states, thereby increasing the average fitness score of the population over time. Since there are 50 vertices, the maximum possible fitness score is 50.

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
    weights ← WEIGHTED-BY(population, fitness)
    population2 ← empty list
    for i = 1 to SIZE(population) do
      parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
      child ← REPRODUCE(parent1, parent2)
      if (small random probability) then child ← MUTATE(child)
      add child to population2
    population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

Figure 1 Pseudocode for the Base Genetic Algorithm

For the vertex cololuring problem, the indviduals of a population will be encoded as a list with each entry being a single letter corresponding to the colour allocated to the vertex of that index. For example, the list *["R", "G", "G"]* means that vertex 0 has been assigned red, vertex 1 has been assigned green, and so on. In accordance to the requirements of part a) of the assignment, we shall restrict the population size to 100 and iterate through 50 generations for the base algorithm.

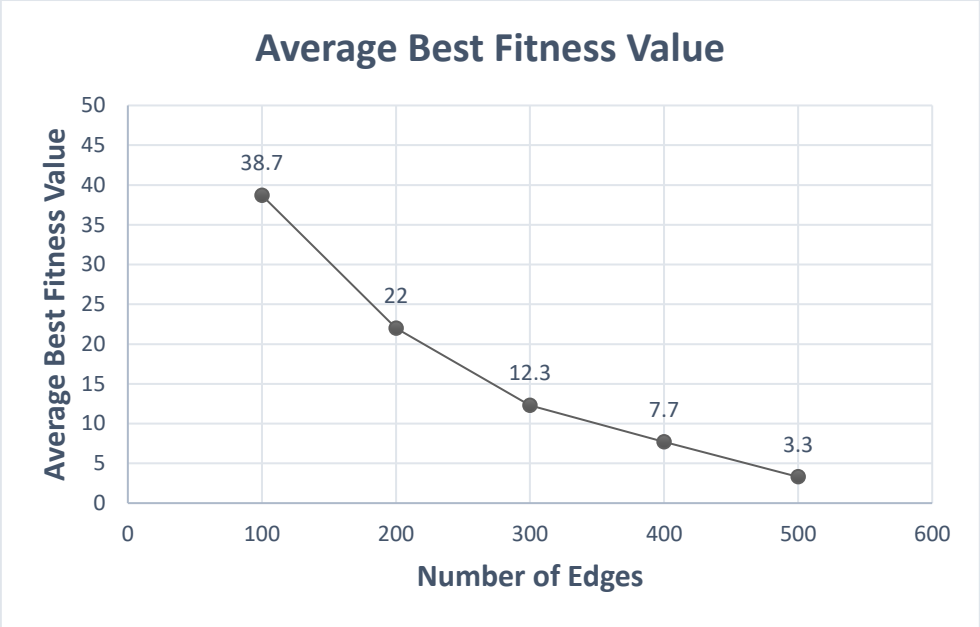

Figure 2 Best Fitness Values averaged over 10 tests

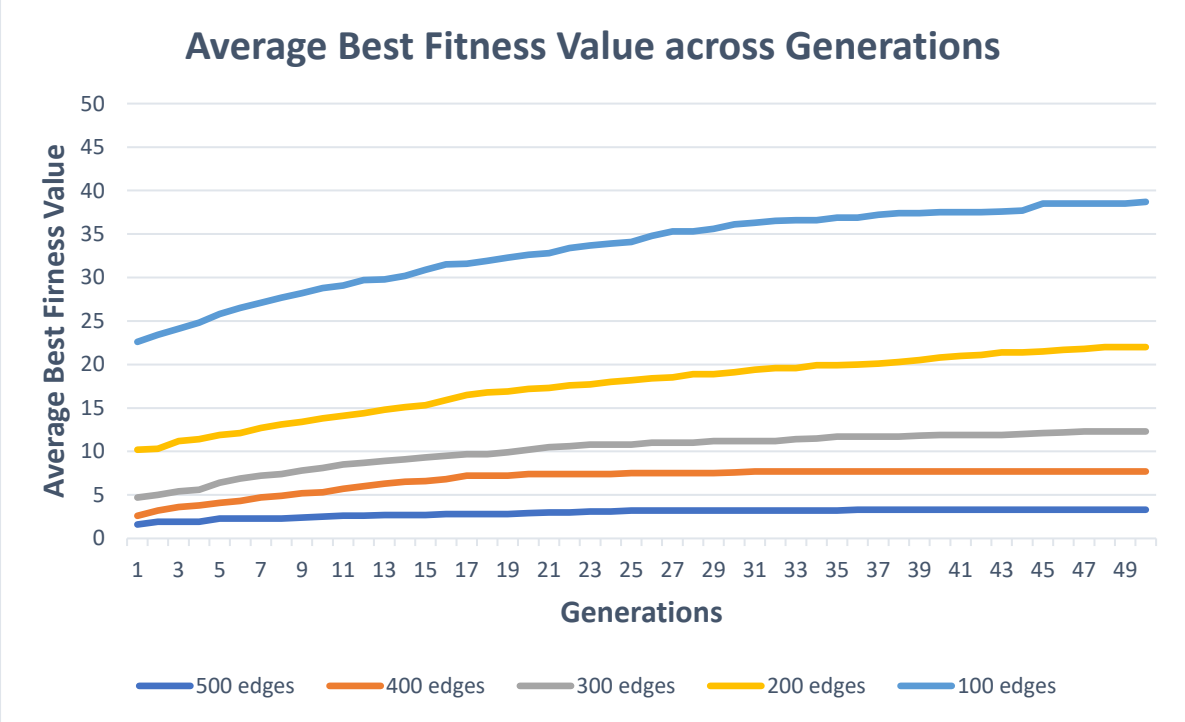

Figure 3 Average Best Fitness Value across Generations, averaged over 10 tests

The results indicate that the base algorithm isn't very effective in successfully finding a high fitness score solution state. Though the results may seem acceptable to some extent for the case of 100 edges, increasing the number of edges sees a drastic drop in the average best fitness value. For the case of 500 edges, it barely makes any progress towards optimality across 50 generations.

There may be multiple factors that explain this inadequacy of the algorithm, some of which we shall address and improve upon in the subsequent hybrid genetic algorithm. However, the primary suspect for this particular case is the number of generations.

For an algorithm that randomly iterates through the search space, 50 generations is simply too low to be able to perform a reasonable extent of exploration. Due to this reason, although the part b) of the assignment suggests that we try and increase the number of generations manually, we shall instead allow the algorithm run for the acceptable duration of 45 seconds before terminating to ensure that it explores as much of the search space as reasonably possible.

A Hybrid Genetic Algorithm

We shall now shift our focus towards improvising the base algorithm. Note that as mentioned above, we've decided to let the algorithm run for as long as possible. Therefore, the number of generations that it took to find the best state will be defined as the first generation in which the best state appeared.

For the sake of keeping this report short, we shall only emphasise on how the average best fitness value changed across the number of edges. Although the assignment asks to record the number of generations it takes to reach the best state, we shall forgo this simply due to the reason that this number varies dramatically and the average fails to provide any useful inferences. For instance, take a look at the figure given below:

78
26
26
191
48
1528

Figure 4 Number of generations to reach maximum fitness value for 50 edges

Ideas and Results

The first modification we make to the algorithm is changing the population size. At first glance, it seems that this change is useful although only by a small margin: the algorithm reaches better average fitness score when the population size increases.

However, this modification comes with a major drawback: in an attempt to increase the exploration space by increasing the population size, the calculation load for every generation increases thereby decreasing the number of generations that can be created within the time limit. When combining with the other modifications we wish to make, the drawbacks outweigh the advantages. For this reason, we choose not to include this in modification in the final improvised algorithm.

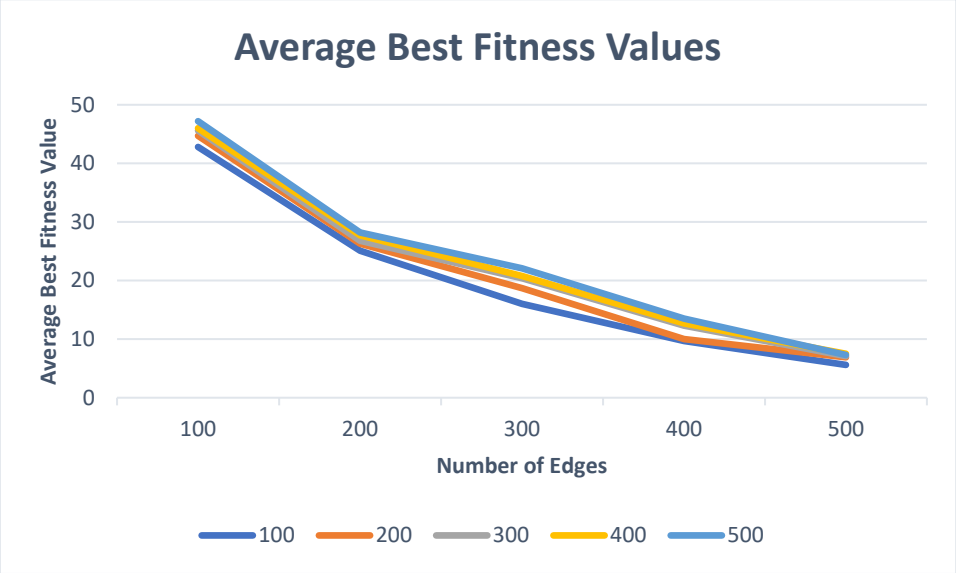

Figure 5 Average Best Fitness Values when Population Size is varied

Next, we move on to modifying the recombination procedure. The textbook implementation uses a single-point crossover, and although there's nothing inherently incorrect about using this variation, it is not the best fit for our case given the nature of the **linkages** between the genes.

Single-point crossover tends to produce better results for cases where genes that are close together also happen to be linked together due to the constraints of the problem (for example, in the N-Queens problem). However, in our case, the linkages are derived from the edges that exist in the graph, wherein vertices whose numbers are consecutive may not always have an edge between them. Thus, we choose uniform crossover as our recombination variation instead, which produces better results.

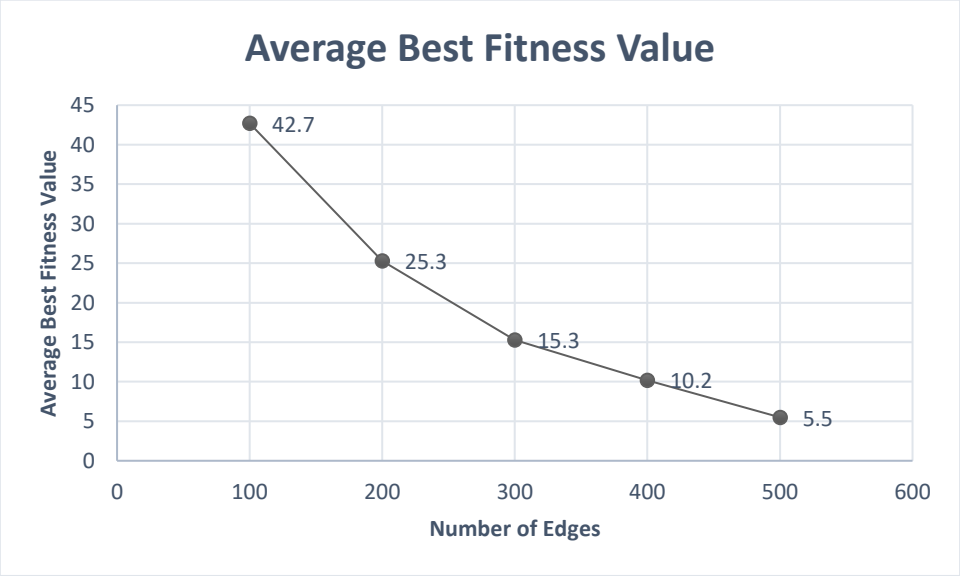

Figure 6 Average Best Fitness Values using Uniform Crossover

We also attempt to modify the mutation procedure in order to induce more useful mutations in the population. For this, we introduce a new term: the **badness** of a node, which we define as the number of adjacent nodes that have the same colour. Naturally, the higher the fitness score for a graph, the lower will be the badness values of its nodes. Thus, by forcing a mutation that lowers the badness, we attempt to increase the overall fitness score of the given graph.

Note that while this process may seem slightly calculation intensive when compared to the original procedure, the mutation probability itself is kept low so as to avoid being a bottleneck for the entire process. Thus, we can safely include this improvisation in the final algorithm without running the risk of reducing the number of generations the process iterates through.

Once again, although the existing procedure is not inherently wrong, the selection procedure for the makeup of the next generation is modified to obtain better results. First, we select the best quarter of the population to pass on to the next generation to retain the high fitness

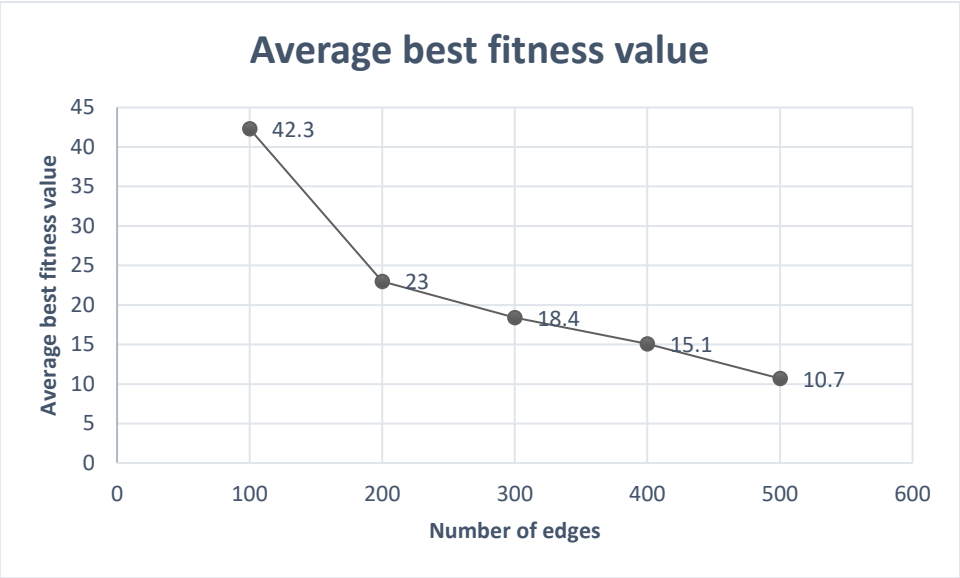

Figure 7 Average Best Fitness Values using "Badness" Bias Mutation

scores we've already achieved. Next, to maintain some variety across the population, we introduce a set of newly created individuals.

Finally, we choose the parents for the children without any fitness score bias. The explanation for the same is simple: two individuals with high fitness scores may not necessarily produce offsprings that share the same attribute. For instance, take the case where two individuals of the population are encoded as *["R", "G", "G", "B"]* and *["B", "R", "R", "G"]*. Since the colours themselves are arbitrary, the two individuals shown above will have the same fitness score. However, this does not guarantee that the children produced by the two will have the same or better fitness score as well, and therefore, in practice, it's beneficial to choose parents without a bias.

Last but not least, we make one final modification to this genetic algorithm: whenever the best fitness scores start to stagnate, we select a fraction of the population and run a hill-climbing algorithm over its individuals in an attempt to increase the fitness score. Although it is a last-ditch effort for improvising the efficiency during runtime, this modification does produce better results though the results may not vary by a huge margin. Again, given that the process is not calculation intensive since it only occurs when the algorithm hits a point of stagnation, we can safely include this in our hybrid genetic algorithm as well.

The results of the genetic algorithm after all of the above modifications have been made are displayed below.

```
(base) PS C:\Users\neeld\Desktop\Assignment1_modified> & C:/Users/neeld/anaconda3/python.exe c:/Users/neeld/Desktop/Assignment1_modified/Submit.py
Roll no : 2020A7PS1223G
Number of edges : 50
Best state :
0:B, 1:B, 2:R, 3:R, 4:B, 5:R, 6:R, 7:G, 8:G, 9:B, 10:R, 11:B, 12:R, 13:B, 14:R, 15:B, 16:G, 17:R, 18:R, 19:R, 20:R, 21:B, 22:G, 23:G, 24:B, 25:G, 26:B, 27:B, 28:G, 29:R, 30:B, 31:G
, 32:B, 33:R, 34:R, 35:R, 36:R, 37:G, 38:G, 39:R, 40:G, 41:B, 42:G, 43:B, 44:R, 45:R, 46:G, 47:B, 48:R, 49:R
Fitness value of best state : 50
Time taken: 0.11997771263122559 seconds
(base) PS C:\Users\neeld\Desktop\Assignment1_modified>
```

Figure 10 Results for the "50.csv" Test File

```
(base) PS C:\Users\neeld\Desktop\Assignment1_modified> & C:/Users/neeld/anaconda3/python.exe c:/Users/neeld/Desktop/Assignment1_modified/Submit.py
Roll no : 2020A7PS1223G
Number of edges : 100
Best state :
0:R, 1:G, 2:B, 3:G, 4:B, 5:B, 6:B, 7:G, 8:B, 9:G, 10:G, 11:B, 12:G, 13:B, 14:G, 15:R, 16:G, 17:G, 18:R, 19:R, 20:B, 21:R, 22:G, 23:R, 24:G, 25:R, 26:G, 27:R, 28:B, 29:G, 30:R, 31:G
, 32:G, 33:R, 34:R, 35:B, 36:B, 37:B, 38:R, 39:G, 40:B, 41:B, 42:B, 43:R, 44:R, 45:G, 46:B, 47:R, 48:G, 49:R
Fitness value of best state : 50
Time taken: 3.4190196990966797 seconds
```

Figure 8 Results for the "100.csv" Test File

```
(base) PS C:\Users\neeld\Desktop\Assignment1_modified> & C:/Users/neeld/anaconda3/python.exe c:/Users/neeld/Desktop/Assignment1_modified/Submit.py
Roll no : 2020A7PS1223G
Number of edges : 200
Best state :
0:B, 1:G, 2:R, 3:G, 4:B, 5:B, 6:R, 7:G, 8:G, 9:R, 10:R, 11:R, 12:R, 13:R, 14:G, 15:G, 16:R, 17:G, 18:B, 19:R, 20:G, 21:R, 22:R, 23:R, 24:G, 25:G, 26:R, 27:R, 28:R, 29:B, 30:B, 31:R
, 32:R, 33:R, 34:R, 35:G, 36:G, 37:G, 38:R, 39:R, 40:B, 41:R, 42:G, 43:G, 44:B, 45:R, 46:G, 47:R, 48:G, 49:R
Fitness value of best state : 30
Time taken: 44.80650186538696 seconds
```

Figure 9 Results for the "200.csv" Test File

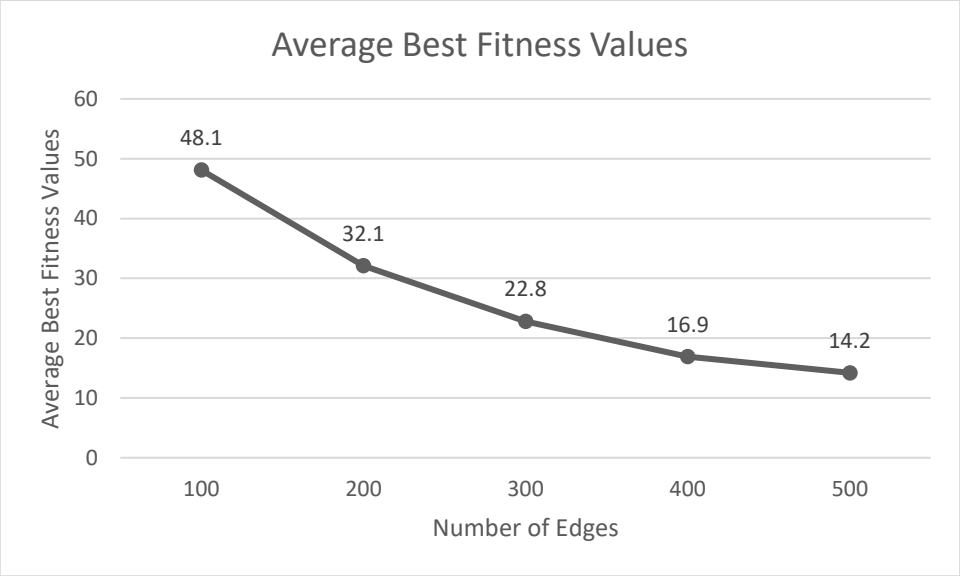

Figure 11 Average Fitness Values for the Hybrid Algorithm