Genetic Coloring - A Genetic Algorithm for Solving the Graph Coloring (NP-Complete) Problem

Soumil Datta
University of Mississippi
University, Mississippi, USA
sdatta2@go.olemiss.edu

Abstract

The graph coloring algorithm is essential to several applications in multiple domains. However, its classification as an NP-complete problem means that there are no found efficient solutions to solve it. While most algorithms depend on the skills, and problem-solving power of the programmer, genetic algorithms have an opportunity to bypass all the human thinking and use randomization to find the closest solution to the problem. In this paper, we introduce Genetic Coloring – a genetic algorithm based graph coloring algorithm. We show that the performance of this algorithm scales well and performs 00x faster than its non-genetic algorithm counterpart.

Keywords: Genetic Algorithm, NP-Complete Problem, Graph Coloring

1. Introduction

Although we are reaching the physical limits of computing performance improvements based on Moore's Law, there is still a class of problems for which efficient solutions are yet to be found. These problems, called NP-complete problems, have been studied rigorously in academic papers. Graph coloring is one such NP-complete problem which does not have any efficient algorithmic solution yet.

Genetic algorithms are becoming increasingly popular in being the algorithms of choice to solve large-sized NP-Complete problems. They are search heuristics that are based on evolution in living organisms. In the real world, actions contributing to a creature's increase in lifespan and survival are awarded by natural selection. Similarly, solutions closer to optimal answer in genetic algorithms are awarded. Genetic algorithms are great at approximating solutions to NP-Hard problems in a reasonable amount of time.

In this paper, we focus on a widely-known NP-Complete problem - The Graph Coloring Problem. We will focus specifically on vertex coloring. In a graph, vertices that are connected to each other by an edge are called adjacent vertices. The goal of this problem is to find the Chromatic Number - the least number of colors such that each pair of adjacent vertices do not have the same color. This paper aims to solve this NP-Complete problem using a Genetic Algorithm and presents findings based on running the algorithm with a combination of different crossover, parent selection, and mutation functions.

2. Algorithm Overview

The genetic algorithm proposed in this paper follows the general structure that is shared between such algorithms, but have some modifications and systems in place to reach the right solution more consistently.

As with any other genetic algorithm, this algorithm performs parent selection, crossovers to produce children, and mutations on those children in a probabilistic way. The fitness is then calculated and then process is repeated over several generations.

The goal of this algorithm is to minimize the fitness. When a fitness of 0 is reached, it means that there exists no connected vertices that have the same color and the graph is therefore colored correctly. However, this does not guarantee the graph utilizing the least number of colors. To ensure this, another mechanism has been put in place.

To ensure the usage of a minimum number of colors, the whole genetic algorithm cycle is repeated from the first generation with with one less color to choose from. In this manner, if the fitness function can reach 0, then the graph can be colored using the current number of colors used or less. Consequentially, if the fitness function never reaches 0, then the genetic algorithm is not able to find an optimal way of coloring the graph with the available colors and the solution is the previous number of colors.

The details of each mechanisms are explained in section 3, and the results shown in section 4.

3. Algorithm Design Details

This section explains the design choices of each important step in my genetic algorithm.

3.1. Graph Representation

The genetic algorithm is run on an graph with *n* vertices. This graph is represented as an n x n adjacency matrix where each rows and columns represent vertices and 1s represent an edge to a vertex. A vertex cannot have an edge to itself, hence the diagonal of the adjacency matrix is forced to contain 0s.

3.2. Chromosome and World Representation

Each chromosome represents a potential solution to the graph coloring problem. The chromosome contains *n* genes - one gene for each vertex in the graph. The genes represent the color of each vertex. Thus the genes can range from 1 to the maximum number of colors needed for the graph. When the chromosomes are newly created, the genes are assigned in random. The maximum number of colors needed for the graph is determined by finding the vertex with the most edges, and summing up the number of edges. The world contains several chromosomes. Upon experimentation and paper reviews, it was found that a world size of 60 is optimal for both performance and correctness. Thus, a world size of 60 is used throughout all experimentation.

3.3. Parent Selection

There are two parent selection methods utilized in this algorithm – Truncation Selection, and Tournament Selection. Truncation Selection picks parents with the lowest fitness scores obtained from the fitness function. This increases the likelihood of their offspring having low fitness scores. Tournament Selection randomly picks a pair of two chromosomes, and selects the fittest of the pair to proceed as a parent.

From my experimentation, it is evident that tournament selection allows for better genetic diversity, and hence allows the algorithm to find a better optimal result. Although truncation selection enables the chromosomes to converge to an approximate solution faster, this solution is often not as great of an approximation as tournament selection.

3.4. Crossover

There are two crossover functions utilized in this algorithm – One-Point Crossover, and Two-Point Crossover. These crossovers result in two children. The reason that it produces two children is to minimize the number of random chromosomes that have to be reintroduced into the population. As long as the number of random chromosomes are minimized, the fittest individuals survive in the population and continue to thrive.

From my experimentation, including random new chromosomes in the population makes the algorithm slower and makes it more difficult for the chromosomes to converge to a solution within the generation bounds. It was also evident that the one-point crossover converges faster than the two-point crossover function. It leads to the algorithm reaching a better approximate solution.

3.5. Mutation

The mutation function mutates child chromosomes based on a probability. In this function, more than one gene is mutated to create a possibility for a more fit chromosome. The mutation function checks if two vertices have the same color in the chromosome. If they do, and if the graph has an edge between those vertices, the gene for the vertex is mutated with a random color. This creates the possibility of the right vertex coloring. The reason for choosing this mutation process was to prevent random mutations from destroying good vertex coloring pairings. This idea was borrowed from the paper by Hindi et al.

There are different mutation rates based on the generation being processed. Before the 200th generation, the mutation rate is high at 65%. Between 200 and 400, the mutation rate drops down to 50%. A mutation rate of only 15% is used for generations over 400.

3.6. Fitness Function

The fitness function assigns a fitness score to the chromosome it is tasked to evaluate. It loops through the genes of a chromosome and compares each gene (vertex) to other genes in the chromosome. In this program, the fitness function must check two important conditions:

1. If the two vertices represented by the genes are connected by an edge in the graph. This can be done by getting the index of each gene and using them to index the location on the adjacency matrix and check if it has a value of 1.

2. If the two vertices represented by the genes have the same color value.

If the above two conditions pass, the pairing is penalized and the fitness value is raised by 1 for each incorrect vertex coloring. This is because two vertices represented by the same color value cannot be connected in the graph.

4. Algorithm Execution Process

The algorithm first generates the graph and finds the max coloring for the graph. It then enters a loop. It creates a new population based on randomly generated chromosomes.

The number of generations to be iterated through is dynamically generated based on the size of the graph. Parent selection is then performed, followed by the crossover to generate two children at a time. This population of new children is then mutated. It is repopulated with random chromosomes if necessary when using truncation selection.

Finally, the fitness is calculated and if there exists a chromosome with a fitness of 0, then the maximum number of available colors is decreased and the process is rerun again. If after the chromosomes do not reach a fitness of 0 after a determined number of generations, then it means that the previous coloring is the best approximate chromatic coloring for the graph.

Since there are often more optimal solutions, the solution is reevaluated for better accuracy. The solution is checked against a non-ga test function based on the implementation from www.codespeedy.com.

5. Performance Evaluation

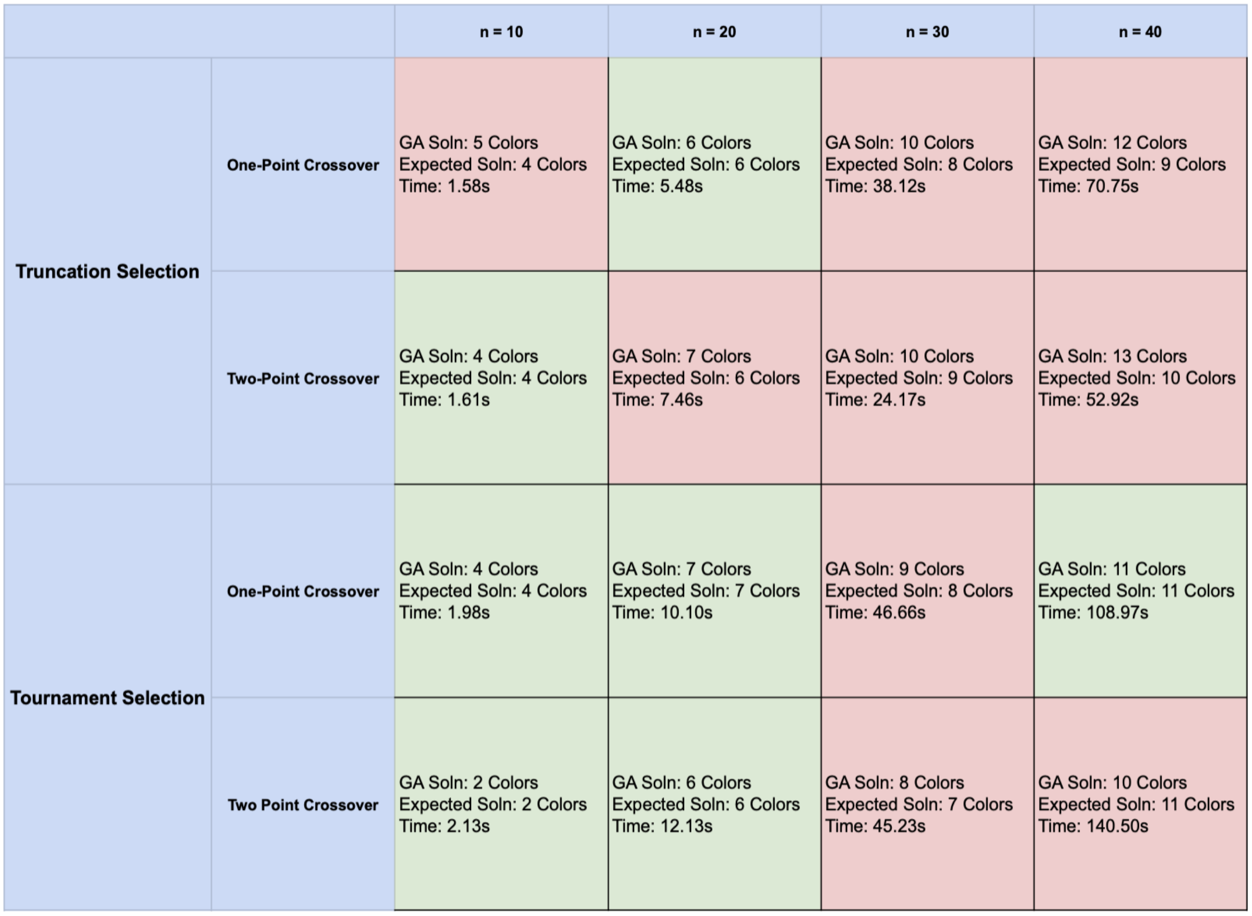
The performance was evaluated with different combinations of the parent selection and crossover methods. These combinations were tested on 5 differently sized graphs (different number of vertices). The results from this experiment are shown in Figure 1. Each of these experiments were run with a population size of 60 and the mutation rates as explained in section 3.5.

6. Conclusion and Future Work

This genetic algorithm seems promising for solving the graph coloring problem. From Figure 1, it is noticeable that Tournament Selection with One-Point Crossover yields the most accurate results. The tuning of probabilities can increase the chance of producing a more optimal solution even more.

Thus, one of the first things to do to generate more optimal solutions is to fine-tune the parameters and randomization that each function goes through. Tuning the rates of mutation and selection could go a long way to produce better solutions.

Additionally, the number of generations that is dynamically assigned based on the size of the graph could be improved since it causes a significant amount of waiting time when checking for a better solution.



		n = 10	n = 20	n = 30	n = 40
Truncation Selection	**One-Point Crossover**	GA Soln: 5 Colors Expected Soln: 4 Colors Time: 1.58s	GA Soln: 6 Colors Expected Soln: 6 Colors Time: 5.48s	GA Soln: 10 Colors Expected Soln: 8 Colors Time: 38.12s	GA Soln: 12 Colors Expected Soln: 9 Colors Time: 70.75s
	Two-Point Crossover	GA Soln: 4 Colors Expected Soln: 4 Colors Time: 1.61s	GA Soln: 7 Colors Expected Soln: 6 Colors Time: 7.46s	GA Soln: 10 Colors Expected Soln: 9 Colors Time: 24.17s	GA Soln: 13 Colors Expected Soln: 10 Colors Time: 52.92s
Tournament Selection	**One-Point Crossover**	GA Soln: 4 Colors Expected Soln: 4 Colors Time: 1.98s	GA Soln: 7 Colors Expected Soln: 7 Colors Time: 10.10s	GA Soln: 9 Colors Expected Soln: 8 Colors Time: 46.66s	GA Soln: 11 Colors Expected Soln: 11 Colors Time: 108.97s
	Two Point Crossover	GA Soln: 2 Colors Expected Soln: 2 Colors Time: 2.13s	GA Soln: 6 Colors Expected Soln: 6 Colors Time: 12.13s	GA Soln: 8 Colors Expected Soln: 7 Colors Time: 45.23s	GA Soln: 10 Colors Expected Soln: 11 Colors Time: 140.50s

Figure 1. Table displaying the GA solution, Expected Solution, and execution time for each combination of selection and crossover functions.