CSCI 446 Artificial Intelligence Project 1 Final Report

Roy Smart   Nevin Leh   Brian Marsh

September 25, 2016

1 Introduction

The *Graph Coloring Problem* (GCP) is the problem of attempting to color a set of interconnected vertices, using a limited set of colors, such that no vertex has the same color as its neighbors. This problem is best visualized as the map coloring problem. As an example of the map coloring problem, consider the problem of assigning one of four colors to every country in the world, such that no two adjacent countries have the same color, as in Figure 1. It can be shown that the map coloring problem reduces to the graph coloring problem[2]

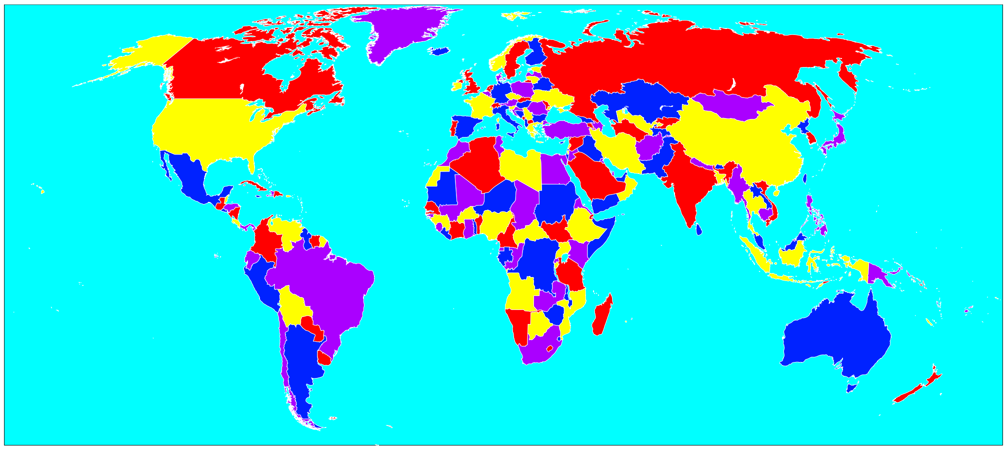


Figure 1: *Map of the world satisfying the map coloring problem [1].*

if we represent the countires as the vertices of the graph, and the borders between countries as the edges of the graph, shown in Figure 2. This configuration produces a *planar* graph, a graph with no edge intersections. For this project, we are asked to solve the GCP problem for planar graphs only. Since the problems are

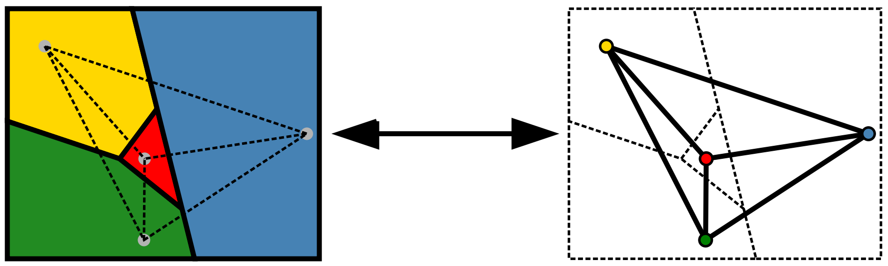


Figure 2: *Diagram describing how the map coloring problem can be transformed into the graph coloring problem [3].*

guaranteed to be planar, we will represent our graphs using polygon maps generated by our program.

To solve the GCP we employed five different algorithms: Minimum Conflicts, Simple Backtracking, Backtracking with Forward Checking, Backtracking with Constraint Propagation (MAC), and Local Search using a Genetic Algorithm. To evaluate these algorithms, we built a problem generating program that can produce a random set of planar graphs. Using the problem generator, we calculated a set of graphs between the sizes {10, 20, 30,...,100} and called the five graph coloring algorithms to solve the GCP. We measured GCP algorithm performance by measuring the number of read/write operations, the time required to find a solution, and the number of function calls for each algorithm. Using these metrics, we predict that the Minimum Conflicts algorithm will be the fastest, based off its performance on the eight-queens problem[2].

The code for this project is implemented in `C++` and depends on the programs `cairo`, `gnuplot`, `ffmpeg`, and LaTeX for graph output, performance output, video output, and documentation respectively.

2 Problem Generation

We created a function that could produce many examples of planar graphs from a set of randomly scattered points. For an arbitrary set of points, there is no unique planar graph that can be constructed. To solve this issue, the problem statement has provided a algorithm for calculating an planar graph.

> Select some point X at random and connect X by a straight line to the nearest point Y such that X is not already connected to Y and line crosses no other line. Repeat the previous step until no more connections are possible.

For our implementation of the above algorithm, we started by creating a complete graph (where each vertex is connected to every other vertex). Each vertex was associated with a list of edges, sorted by length. We then selected a point at random and inspected the first unchecked edge E. If E did not cross any of the accepted edges, it is added to the list of accepted edges and marked as checked. This process was repeated until every edge was marked as checked.

The most challenging part of the implementation described above was determining if two line segments intersected. An algorithm to determine if two edges (or line segments) cross has been outlined by LaMothe [4] and described in detail here. Let's start by defining two line segments

$$\begin{aligned} A &= \{\mathbf{a},\ \mathbf{a'}\} \\ B &= \{\mathbf{b},\ \mathbf{b'}\} \end{aligned} \tag{1}$$

where $\mathbf{a}$, $\mathbf{a'}$, $\mathbf{b}$, and $\mathbf{b'}$ are vectors from the origin to the ends of the line segments. Next, compute the direction vectors for each line segment

$$\begin{aligned} \boldsymbol{\alpha} &= \mathbf{a}' - \mathbf{a} \\ \boldsymbol{\beta} &= \mathbf{b}' - \mathbf{b} \end{aligned} \tag{2}$$

Now, the trick to solving this problem is to parameterize the line segment using the direction vector and a parameter t_i in the domain [0, 1].

$$\begin{aligned} \mathbf{p} &= \mathbf{a} + \boldsymbol{\alpha} t_a, \quad t_a \in [0,\ 1] \\ \mathbf{q} &= \mathbf{b} + \boldsymbol{\beta} t_b, \quad t_b \in [0,\ 1] \end{aligned} \tag{3}$$

From here, the solution is obvious. To find the point of intersection, we set $\mathbf{p} = \mathbf{q}$ and solve for the values of t_a and t_b. Then, if the solution is outside the domain of t_i, the line segments do not intersect. Thus,

$$\Rightarrow \begin{cases} p_x = q_x \\ p_y = q_y \end{cases} \tag{4}$$

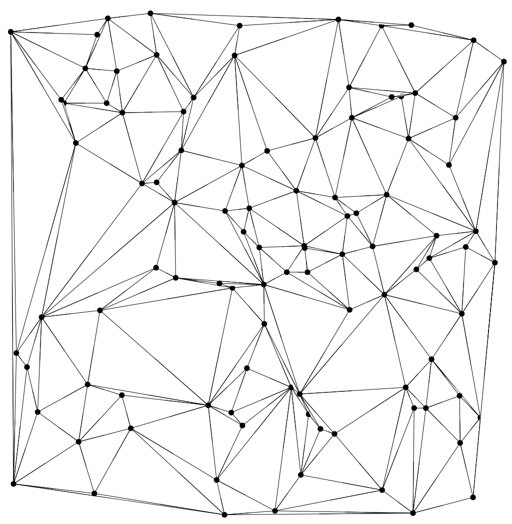
$$\Rightarrow \begin{cases} a_x + \alpha_x t_a = b_x + \beta_x t_b \\ a_y + \alpha_y t_a = b_y + \beta_y t_b. \end{cases} \tag{5}$$

Equation 5 is a system of two equations and two unknowns, solving for t_a and t_b gives

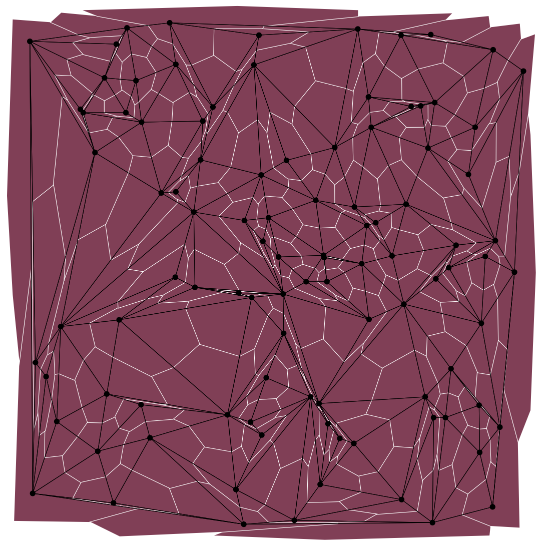
$$\begin{aligned} t_a &= \frac{(-a_y + b_y)\beta_x + (a_x - b_x)\beta_y}{\alpha_y\beta_x - \alpha_x\beta_y} \\ t_b &= \frac{(a_y - b_y)\alpha_x + (-a_x + b_x)\alpha_y}{-\alpha_y\beta_x + \alpha_x\beta_y}. \end{aligned} \tag{6}$$

Finally, we can determine whether A and B intersect if $0 < t_a < 1$ and $0 < t_b < 1$ evaluates to true. This method may be more inefficient than other algorithms, such as the example outlined by Cormen[5] due to the division operation in Equation 6.

Using the expression above to determine line intersections, we were able to implement a problem generator that was reasonably efficient, generating approximately 1×10^3 examples per second. This efficiency was important, because it made testing and debugging the GCP algorithms much faster. Example output from the problem generator is shown in Figure 3a.



(a) *Planar graph with 100 vertices.*



(b) *Polygon map corresponding to (a).*

Figure 3: *Visualization of the output of the problem generator. First a planar graph was created (a), and then a corresponding polygon map was found (b).*

We have also implemented a way to produce a map of regions that is analogous to the graphs created by the problem generator, shown in Figure 3b. Since the graph coloring problem does not uniquely define the map coloring problem, we had to invent a metric that produces appropriate graphs. To define our metric, consider that if a polygon G_i corresponding to each point P_i were constructed out of the midpoints $M_{i,m}$ of the associated edges $E_{i,m}$ of P_i, then an adjacent polygon G_j would only touch at the corners, that is, our map would be full of gaps. Therefore, to fix the problem we also define the centroid $C_{i,n}$ of each triangle $T_{i,n}$ formed by the two adjacent edges $E_{i,m}$ and $E_{i,m+1}$. We then define the final polygon for P_i using the points $M_{i,m}$ and $C_{i,n}$. This procedure is unnecessary in terms of the assignment, but is helpful for debugging GCP algorithms, demonstrating example output, and for aesthetic pleasure.

3 Minimum Conflicts

4 Simple Backtracking

4.1 Description

Backtracking is a method of solving constraint satisfaction problems through an "educated" exhaustive search. It works by building a partial solution one component at a time and then testing the solution to check if the solution still has a chance of success [6]. If the solution cannot possibly succeed, the algorithm *backtracks* by moving back one component, modifying the component to form a new partial solution and then continuing. By checking partial solutions, backtracking can eliminate large portions of the solution-space, thus increasing the efficiency of the search.

In the graph coloring problem, backtracking assigns colors to one vertex at a time and checks to see if it has the same value as any of its neighbors. If none of the possible colors work for the current vertex, it backs up and tries a different color on a previously assigned vertex. If a color has no conflicts, backtracking continues to the next vertex in the sequence.

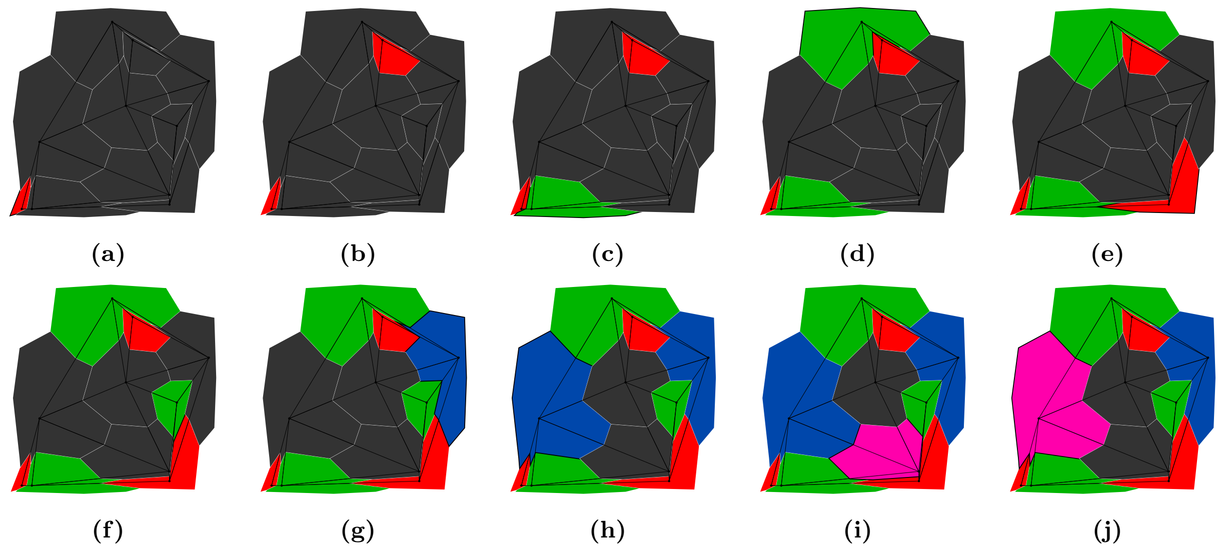


Figure 4: *A map coloring example using simple backtracking on an N=10 graph up to the first backtrack event. In panels (a) through (i), each vertex is assigned the first available color in the order (red, green, blue, pink). Once the first pink assignment is made in panel (i), there are no possibilities remaining for the final vertex. The algorithm then backtracks and changes the node assigned in (h) to pink and tries again.*

To implement simple backtracking we followed the recursive algorithm outlined in Russel and Norvig Section 6.3 [2]. Russel's Backtrack loops through each value in the domain, and if the value is consistent with the constraints it recursively calls Backtrack. The function returns `true` when every variable has been assigned, and `false` if it cannot satisfy the constraints. The recursive function is advantageous, as it saves the state of the stack before each recursive call, making it trivial to restore the state of the solution after a failed recursive call to backtracking.

The crux of the backtracking procedure in the graph coloring problem is checking whether a coloring is consistent with constraints. From profiling our code, it was obvious that a fast constraint checking function was critical to the ability of backtracking to solve graph coloring problems up to $N = 100$ vertices in a reasonable amount of time. This facilitated a change of design from an object-oriented approach, to an array-based approach.

4.2 Experimental Approach

To evaluate backtracking's performance on the graph coloring problem, we ran the algorithm on ten graph sizes $N = (10,\ 20,\ ...,\ 100)$ with 20 trials for each value, for a total of 200 backtracking experiments. From each experiment we recorded the number of vertex reads, vertex writes, and number of recursive calls. Time constraints on this project demanded that we set a limit on the number of calls to backtracking, therefore we also recorded if this limit was reached for each experiment.

We expect that the number of recursive calls and the number of vertex writes and the number of vertex reads to be correlated by some constant factor. This is because there is a maximum of two writes and N reads for each recursive call. Therefore, we can use this information to verify that the algorithm is working properly.

4.3 Results

Figure 5: *Logarithmic plot describing the performance of the backtracking algorithm vs. the number of vertices. The shaded regions represent the minimum and maximum values of each quantity.*

5 Backtracking with Forward Checking

5.1 Description

Backtracking with forward checking incorporates the usual simple backtracking algorithm described in the preceding section, while incorporating an ability to look into the future to see if a particular assignment would create an inconsistency. Forward checking achieves this by trying to make an inconsistency occur as fast as possible by checking if all current assignments are consistent with each other [7].

To accomplish forward checking in the graph coloring problem, we need to modify our representation of the graph to allow superpositions of colors and initialize the graph such that all colors are possible for each vertex. Next, for each call to backtracking, we assign a color to the next vertex, and delete that color from the vertex's neighbors list of possible colors. If any vertex has its list of possible colors completely eliminated, the algorithm backtracks and changes a color further up the tree.[2].

5.2 Experimental Approach

5.3 Results

6 Backtracking with Constraint Propagation

6.1 Description

6.2 Experimental Approach

6.3 Results

7 Genetic Algorithm

7.1 Implementation

Local search using a genetic algorithm involves a process similar to evolution to produce an individual that maximizes a fitness function. Since genetic algorithms are an abstraction of biological evolution, they involve

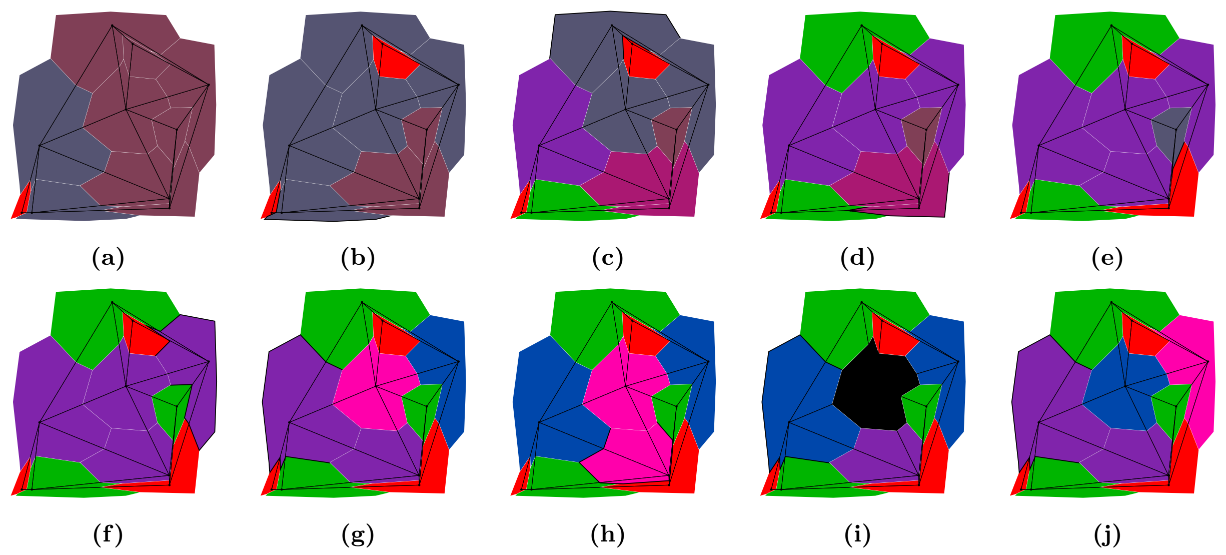


Figure 6: *The map coloring example from Figure 4 using backtracking with forward checking on an N=10 graph up to the first backtrack event. In panels (a) through (g), each vertex is assigned the first available color in the order (red, green, blue, pink), and deletes that color from the possible color of its neighbors. In panel (h) blue is assigned to the node on the left side, leaving two of its neighbors with only pink in their domain. Next, in panel (h), the central vertex has its domain reduced to the empty set, prompting a backtrack to panel (j). Note that forward checking in this case did not provide any advantage over backtracking.*

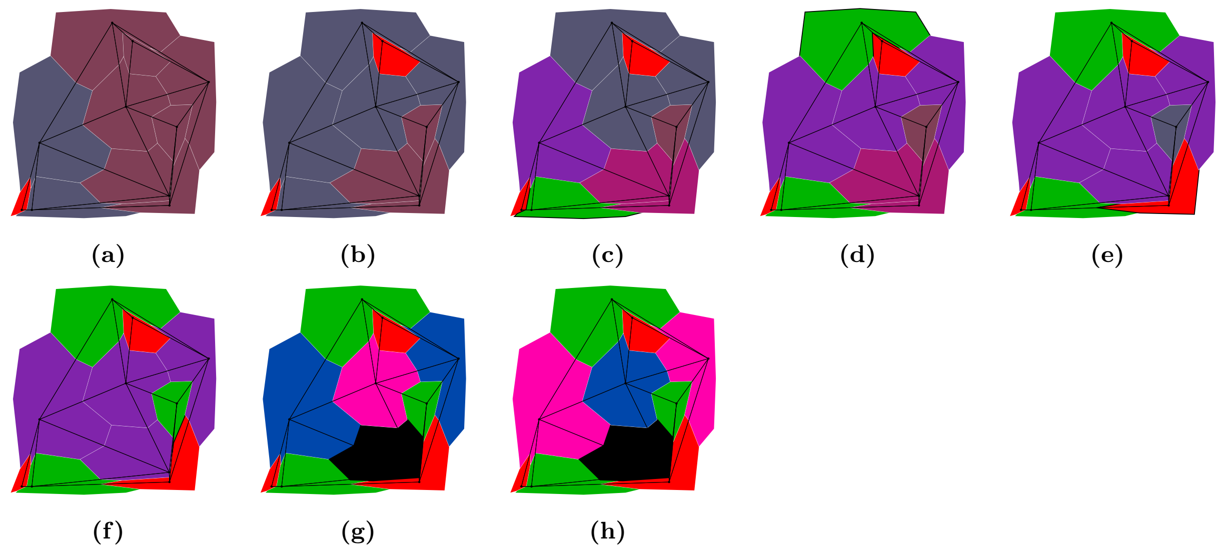


Figure 7

some of the classic mechanisms seen in the natural world such as crossover, mutation, and survival of the fittest[8].

The genetic algorithm begins with a population of randomly generated individuals. A selection process chooses which individuals to use for reproduction, which are weighted by their score generated the fitness function. The individuals are recombined by cross-over and the resulting individuals may be mutated depending on a small probability.

To apply the genetic algorithm to the GCP we started with the genetic algorithm described in Russel and Norvig Section 4.1[2] and implemented each feature in accordance with the GCP. Our initial population consisted of N randomly colored graphs. It was hard to determine what population size was the best, but it was clear that the optimal population size was related to the size of each graph and that larger graphs needed larger populations in general. N was a natural number to try and it seemed to work fairly well.

To select individuals for crossover we used tournament selection where two individuals were randomly selected and the one with the best fitness score proceeded to crossover. Naturally, two tournaments were needed to get the two individuals required for crossover. In this scheme an individual could be chosen zero, one, or multiple times to participate in tournaments.

The fitness function used to calculate the fitness score was essentially a penalty function because only the total number of conflicts in each individual was used to calculate the fitness score. Since this score reflected how many neighboring vertices shared colors, a large score was considered less fit and a score of zero indicated that a solution was found.

The crossover algorithm consisted of random single point crossover. To achieve this we chose a random point in each graph and swapped colors up to that point. The crossover example in Figure 8 looks like multi-point crossover because the vertices of the graph are unordered when crossover is performed.

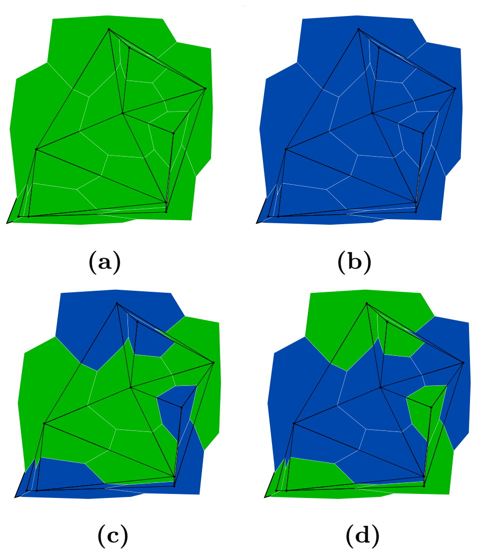


Figure 8: *Simple example of crossover where (a) and (b) are the parents and (c) and (d) are the offspring.*

After crossover we immediately mutated each individual which consisted of selecting each vertex and randomly changing it depending on a mutation rate defined as $1/N$. This mutation rate was chosen because it seemed to minimize the number of iterations to find a solution. This number proved to be very specific and any deviation would negatively affect the performance of the algorithm considerable.

Once we had the new individuals we inserted them into the new population. We used pure insertion meaning that each generation was composed entirely of new individuals produced by crossover. This was later understood to be a poor choice, but it did work for this application.

This process of selection, crossover, mutation, and reinsertion was repeated until an individual with zero conflicts was found or a predefined limit on the number of generations allowed was reached. In Figure 9 a flow graph showing the general flow of our genetic algorithm. It can be seen that the population is initialized outside of the main loop.

7.2 Experimental Approach

To gauge the performance of the algorithm we will run the algorithm on graph sizes $N = (10, 20, ..., 100)$ with 20 runs of each different graph size. The metrics that will be used are number of generations, number of writes, and number of reads.

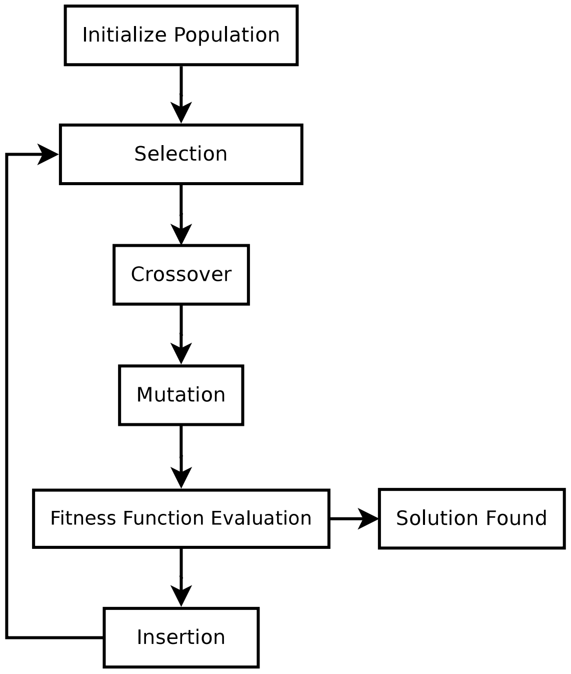


Figure 9: *Flow graph of our genetic algorithm.*

The most obvious metric to use was number of generations because it tells you how many steps it took to find a solution. It was used to see how performance changed as N increased and to compare the relative difficulty of graphs of the same size. Another added benefit of this metric was that it was used to find the optimal population size and mutation rate. The drawback of this metric was that it did not take into account the increased computation needed for larger graphs.

One way we accounted for the increased computation time needed for larger graphs was to count the number of times we check the color of a graph vertice. This scaled with the size of the graph for two reasons. One was that each time we checked the fitness of a graph we had to read each nodes color and, as the graph got bigger, more nodes had to be read. The other reason is that, since our population is of size N, we have to check more graphs as the population size grows. We called this metric `num_reads`

The other way we accounted for the increased computation time needed for larger graphs was to count the number of times we change the color of a graph vertice. This metric scaled with the size of the graph for the same reasons as number of reads.

7.3 Results

Figure 10 shows our metrics on a logarithmic plot. There are several points of interest on this graph. The first is how the number of generations increase as N increases. We can deduce from the slope of the lines that this algorithm runs in exponential time, thought it is still much faster than a brute force approach. We can further deduce that the number of generations increases at $\mathcal{O}(e^{Na})$ where a represents the slope of the line in a logarithmic plot. By deduction we can see that the slope and therefore $a < 1$.

The other points of interest are the number of reads and writes. It can be seen that the slope of the line is a little more steep than the number of generations. This is as expected because the population size increases as N increases and therefore causes the number of reads and writes to grow. Even with these setbacks read and write still managed to run in approximately $\mathcal{O}(e^{Nb})$ where b represents the slope of the line in a logarithmic plot. By deduction we can see that the slope is greater than that of the number of generations so $a < b$. Additionally we can see that the slope is less than one so we can further constrict b to $a < b < 1$.

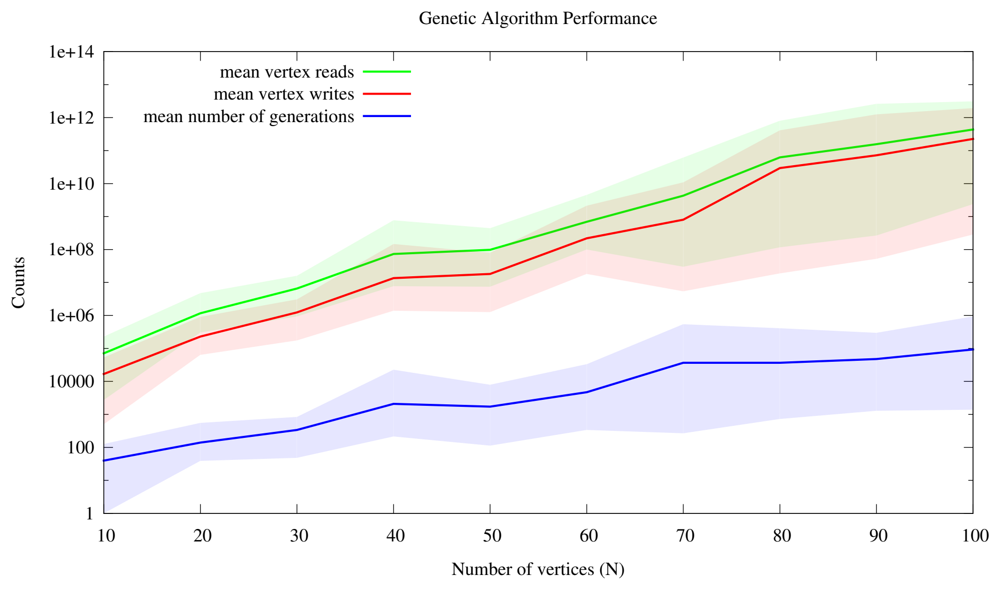


Figure 10: *Logarithmic plot describing the performance of the genetic algorithm vs. the number of vertices. The shaded regions represent the minimum and maximum values of each quantity.*

8 Comparing Algorithm Performance

8.1 Experimental Approach

8.2 Results

9 Summary

References

[1] Wikipedia, the free encyclopedia. Four color theorem, 2016. `https://upload.wikimedia.org/wikipedia/commons/4/4a/World_map_with_four_colours.svg`.

[2] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, Upper Saddle River, New Jersey 07458, 3rd edition, 2010.

[3] Wikipedia, the free encyclopedia. Four color theorem, 2016. `https://en.wikipedia.org/wiki/Four_color_theorem#/media/File:Four_Colour_Planar_Graph.svg`.

[4] André LaMothe. *Tricks of the 3D Game Programming Gurus.* Sams Publishing, 201 West 103rd Street, Indianapolis, Indiana 46290, 2003.

[5] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms.* Massachusetts Institute of Technology, Cambridge, Massachusetts, 3rd edition, 2009.

[6] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, October 1965. `http://doi.acm.org/10.1145/321296.321300`.

[7] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263 – 313, 1980. `http://www.sciencedirect.com/science/article/pii/000437028090051X`.

[8] Prof. S.V. Chande and Dr. M. Sinha. Genetic algorithm: A versatile optimization tool. *BVICAM's International Journal of Information Technology*, 2008.