CSCI 446 Artificial Intelligence Project 1 Final Report

Roy Smart Nevin Leh Brian Marsh

September 26, 2016

Abstract

abstract-text

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]

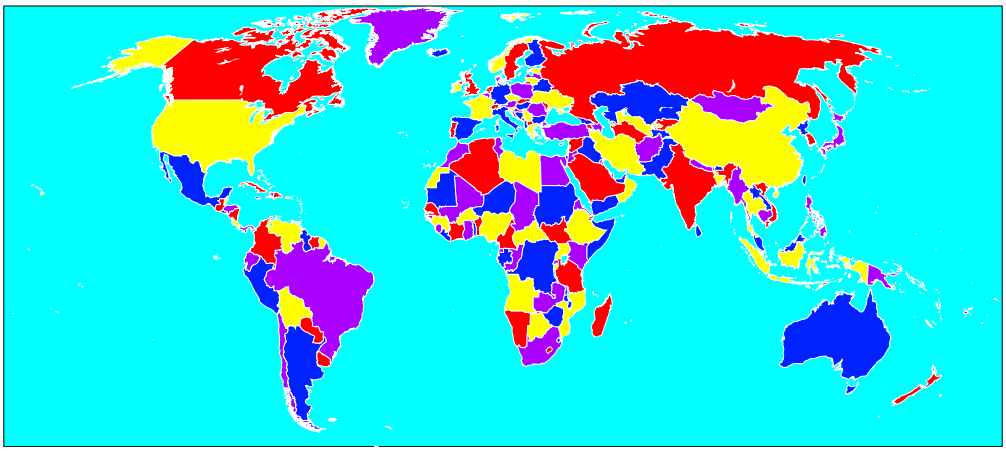


Figure 1: *Map of the world satisfying the map coloring problem [1].*

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 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[4].

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.

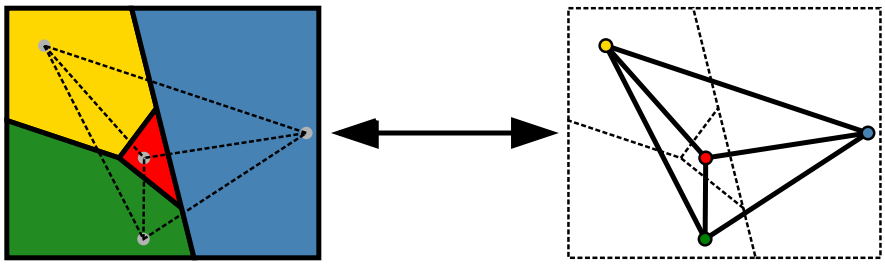


Figure 2: *Diagram describing how the map coloring problem can be transformed into the graph coloring problem [3].*

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 [5] 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,

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

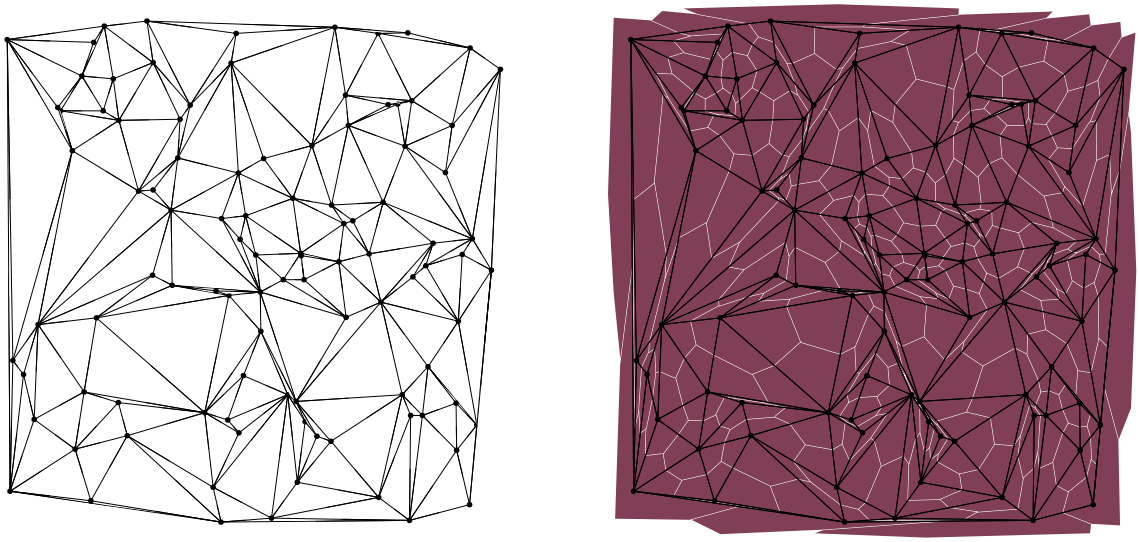
$$\Rightarrow \begin{cases} a_x + \alpha_x t_a = b_x + \beta_x t_b \\ a_y + \alpha_y t_a = b_y + \beta_y t_b. \end{cases} \tag{5}$$

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[6] 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.

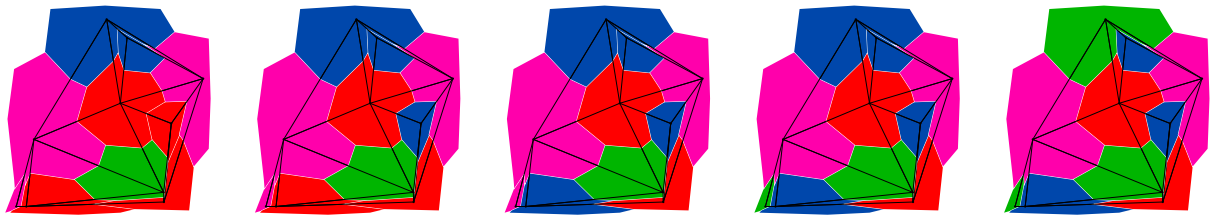
3 Minimum Conflicts

3.1 Description

Minimum conflicts uses a relatively simple algorithm to solve constraint satisfaction problems. A step towards solving the problem is made based upon how many conflicts a variable has with other variables. As it runs,

the algorithm updates the number of conflicts until none remain for any of the variables, meaning the problem is solved, or a time limit is reached, in which no solution is found.

The implementation of minimum conflicts for the graph coloring problem begins by coloring the graph at random. Each node of the graph is given a conflict score to represent how many conflicts it has with its neighbors. The algorithm starts by randomly selecting a node with conflicts and changing it to a color that will result in the minimal conflict score. The affected nodes update their conflict scores and the process repeats until no conflicts remain.



(a) **(b)** **(c)** **(d)** **(e)**

Figure 4: *A map coloring example using minimum conflicts on a graph with 10 vertices.*

Figure 4 provides an example of minimum conflicts to better describe the steps it takes. The first image is a randomly colored graph in which $N = 8$. An easy to see example is the transition from 4d to 4e. The algorithm randomly selects a node with conflicts, the blue node at the top of the panel, and considers all of the possible colors it could be changed to. This node has a conflict because it borders another blue node. The color with the lowest conflict score, green, is selected and the node is changed to this color. Green was selected because the node already bordered a red, pink, and red region. The result is a correctly colored graph. If there were more conflicts, the process would repeat until the graph did not have any more conflicts.

Our implementation of the minimum conflicts algorithm is based upon Figure 6.8 in Russel and Norvig [2] which outlines the structure of a minimum conflicts algorithm. The result was a `for` loop that ends upon a maximum number of steps being reached, unless a solution is found first. The function returns `true` if it ends prematurely from the check `if (map->num_conflicts() == 0)` is satisfied, which indicates a solution. Otherwise, the function returns `false` if the maximum number of steps is reached and the `for` loop is completed, indicating that a solution was not found. At each iteration of the loop a conflicting node is chosen at random and is minimized.

One important feature of our implementation was that, if every color caused the same number of conflicts, a random color was chosen for that vertex rather than keeping its original color. This kept the program from getting deadlocked in a situation where one vertex was depending on a second vertex that relies on a third vertex. If the third vertex is chosen but remains the same color it can keep the second vertex from changing color and, therefore, the first vertex is stuck.

3.2 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. From each run we saved the number of vertex reads, vertex writes, steps taken by the program. The number of steps was defined as the number of times a vertex was chosen to be minimized. This metric was useful for measuring backtracking against itself while the vertex reads and writes were good for comparing its performance to the other algorithms.

3.3 Results

Figure 5 shows that minimum conflicts ran in exponential time. While the lines are linear on the plot, the plot is logarithmic, implying that the exponent is increasing by some constant multiplied by N. An interesting point is that even though the algorithm is running in exponential time, it is much faster than brute force and

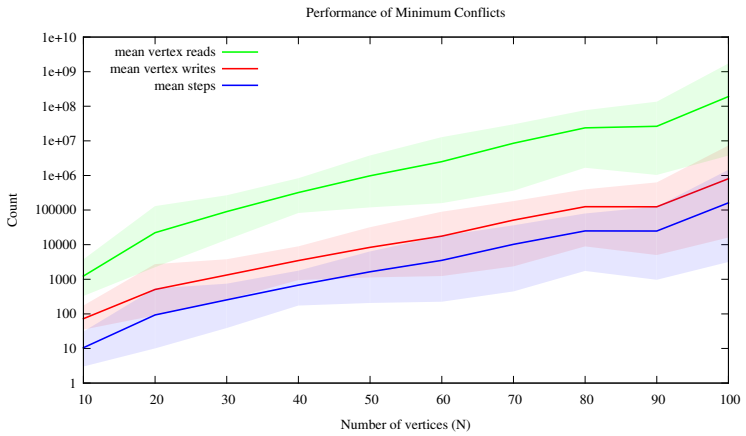


Figure 5: *Logarithmic plot describing the performance of the min-conflicts algorithm vs. the number of vertices. The shaded regions represent the minimum and maximum values of each quantity.*

faster than any of the other algorithms tested. This follows our expectations that stemmed from the fact that minimum conflicts is faster than backtracking when solving the eight-queens problem[4].

There are several interesting aspects of the plot that can be explained by the structure of our code. The first is that the number of vertex reads far outstrips number of steps and writes. This is due to the fact that in each step all of the chosen vertex's neighbors are read while only the vertex itself is written to.

Another interesting aspect is that the number of vertex writes slightly exceeds the number of steps taken. This is explained by the fact that the algorithm actually has to write a color each time it checks that color for number of conflicts. Since the algorithm has to check four colors and then write a color the total number of writes per step is actually five.

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 [7]. 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 yielding greater efficiency than a simple exhaustive 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.

To illustrate the process of backtracking consider the sequence of images in Figure 6. This sequence follows the backtracking algorithm assigning the first acceptable color (in the order red, green, pink) to each vertex until the first backtracking event in panel 6j. The colors assigned to the vertices (a) through (g) seem reasonable until blue is assigned to vertex (h) in panel 6h. Then in panel 6i we can see that the only acceptable color for vertex (i) is pink and even worse, vertex (j) has cannot be assigned a color without being in conflict with its neighbors. Therefore, in panel 6j the algorithm is forced to backtrack to vertex (h) and try pink (which we will see).

The example in Figure 6 seeks to describe the major issue with backtracking: it can get stuck in situations where it can take an inordinate amount of time to backtrack far enough to fix the problem. In panel (j), after

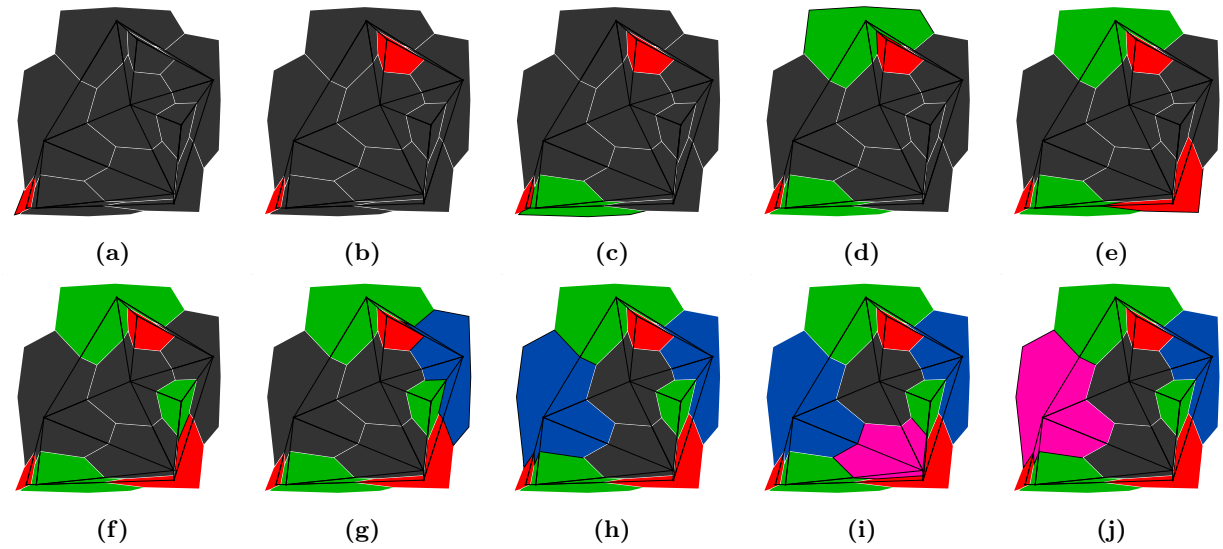


Figure 6: *A map coloring example using simple backtracking on a graph with 10 vertices. From this diagram we will define a label for each vertex by the panel during which it is assigned, for example vertex (a) is colored red in panel 6a, vertex (c) is colored green in the panel 6c, etc. Note we will define vertex (j) as the last uncolored node, since panel 6j changes the value on vertex (h).*

the backtracking operation, we can see that the algorithm did not retreat far enough, changing vertex (g) to pink still results in a conflict in vertex (j). The algorithm needs to back up to vertex (e) and change it to blue to resolve the conflict on vertex (j) and solve the graph. We will see in the following sections on forward checking and constraint propagation if we can mitigate this problem.

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

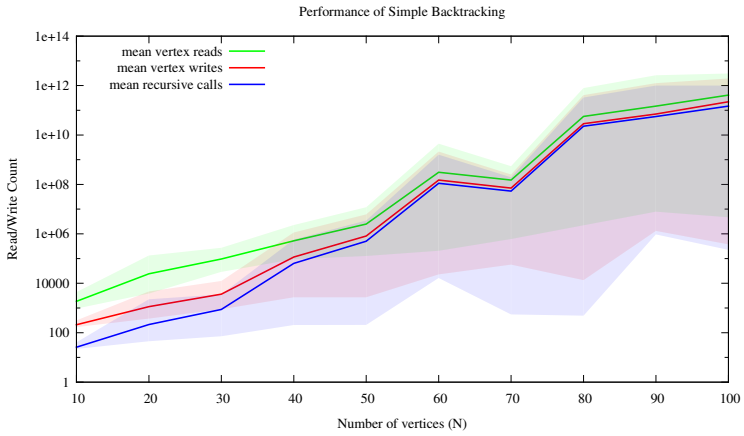


Figure 7: *Logarithmic plot describing the performance of the backtracking algorithm vs. the number of vertices. The shaded regions represent the minimum and maximum values of each quantity.*

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 [8].

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].

To visualize how backtracking with forward checking is different from simple backtracking, we will consider again the map coloring example from the preceding section. To represent the superposition of colors required by this algorithm, we have formed a color map that represents additive mixings of the four pure colors. In this scheme, the color with all colors activated is mauve [RGBP]. The example starts with the graph initialized to mauve. Then vertex (a) is set to red, deleting red from its neighbors, vertices (h) and (c), forming gray [GBP]. After a similar process for vertex (b), vertex (c) is assigned green, deleting green from vertices (e) and (i) to form magenta [RBP] and from vertex (h) to form purple [RP].

This process produces similar results to simple backtracking until vertex (g) is colored blue in panel 8g, it then deletes blue from vertex (j), leaving pink. Next, in panel 8h, vertex (h) is chosen to be blue, leaving vertex (i) with pink as its only possible color. In panel 8i the algorithm selects the only choice for vertex (i) (pink), leaving vertex (j) with the null set. Since the null set is not allowed, the algorithm backtracks to vertex (g) in panel 8j, and changes its color to pink.

In this example, forward checking had no advantages over simple backtracking, as it discovered the inconsistency in vertex (j) in approximately the same number of steps. However, forward checking offers immense advantages for large graphs, where it can check if a color is consistent with its neighbors much sooner than simple backtracking, leading to much faster solutions.

For our implementation, we developed a function denoted `forward_check(vertex)` that checks the color of `vertex` and deletes that color from any neighboring vertices. If any neighboring vertex does not have any

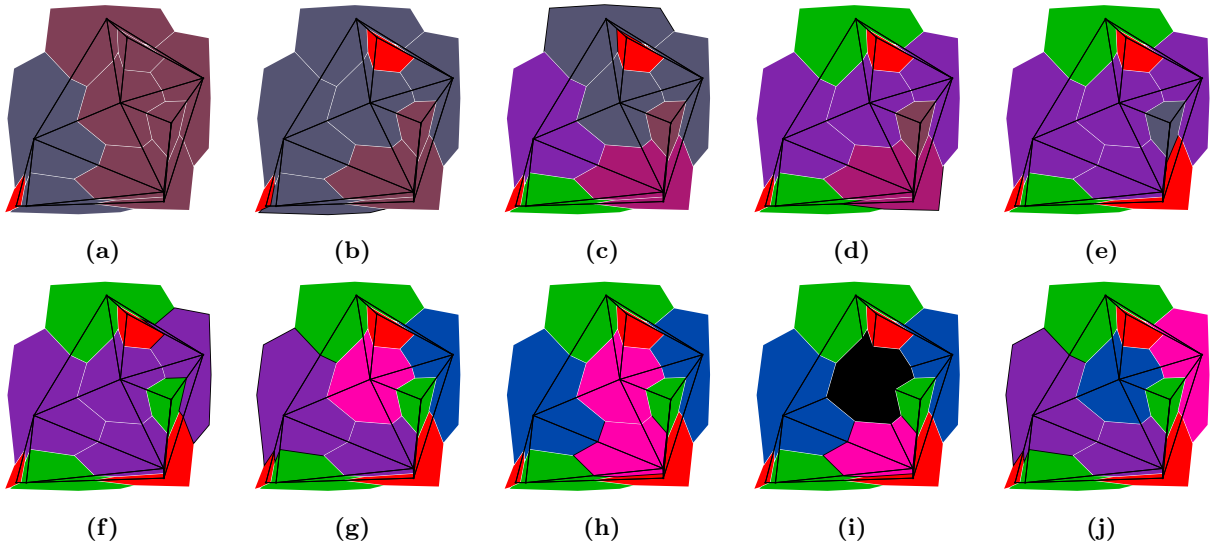


Figure 8: *The map coloring example from Figure 6 using backtracking with forward checking. Note that the intermediate colors represent additive mixings of the four pure colors (red, green, blue, and pink), for example purple (the color of vertex (h) during panel (c)) is a combination of blue and pink, magenta is a combination of red and blue, etc.*

possible colors left, `forward_check()` returns `false` otherwise it returns `true`. This function is called by the same BACKTRACK function described in Section 4.1, with the additional instruction that `forward_check()` must evaluate to `true` before the next recursive call to BACKTRACK.

5.2 Experimental Approach

Our experimental approach for this algorithm is essentially the same as the approach for simple backtracking described in Section 4.2. We will still be measuring the number of vertex reads, vertex writes, and the number of recursive calls. However, since the forward checking algorithm must carry out more decisions for each recursive call, we expect there to be a larger amount of vertex reads and vertex writes for each recursive call. We will see in Section 8 if this additional overhead was worth it compared to simple backtracking. As with simple backtracking there is also a limit on how long each experiment can run, but this ends up not being important, as

5.3 Results

6 BACKTRACKING WITH CONSTRAINT PROPAGATION

6.1 Description

Backtracking with constraint propagation uses the idea of *maintaining arc consistency* (MAC) to try to improve upon the performance of backtracking with forward checking. This variation of the backtracking algorithm can be described as recursive forward checking [2]. Where backtracking with forward checking assigns a color and deletes that color possibility from its neighbors, backtracking with constraint propagation assigns a color, deletes that color from its neighbors, and then uses that information to delete additional color possibilities from its neighbor's neighbors. This is the idea of constraint propagation. Furthermore, this algorithm makes sure to maintain arc-consistency by making sure that every vertex retains a color that is consistent with the rest of the graph.

To see how backtracking with constraint propagation improves upon the idea of backtracking with forward

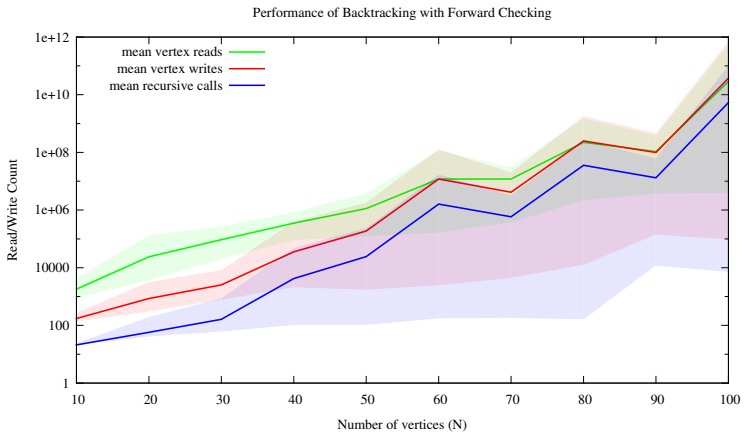


Figure 9: *Logarithmic plot describing the performance of the backtracking algorithm vs. the number of vertices. The shaded regions represent the minimum and maximum values of each quantity.*

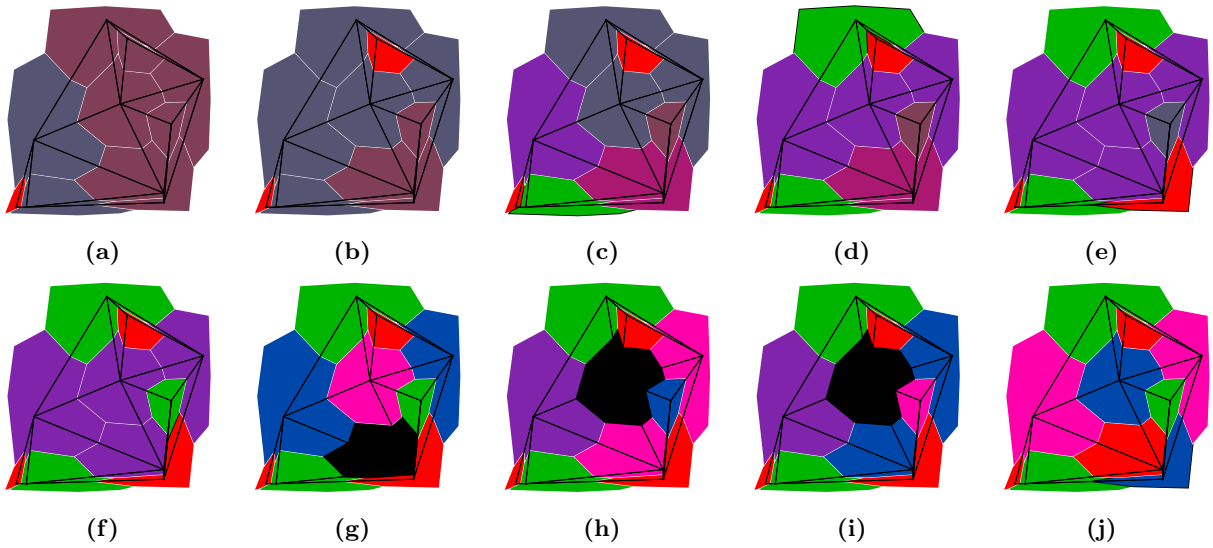


Figure 10: *The map coloring example from Figure 6 using backtracking with constraint propagation.*

checking, we turn one last time to the example introduced in Section 4.1. Start by comparing Figures 8 and 10. Notice how panels up to 8f and 10f are precisely the same. Backtracking with forward checking and backtracking with constraint propagation haven't diverged up to this point since this is a small graph. The first change occurs when vertex (g) is assigned the color blue in panel 10g. In the forward checking example this reduces the color possibilities on vertex (j) to pink in panel 8g. But in the constraint propagation example it propagates this information to vertex (h) and (i) in panel 10g, reducing them to blue and null, respectively. Since constraint propagation noticed this error two steps earlier than forward checking, it is able to quickly recurse back in panels 10h and 10i far enough to change vertex (e) to blue in panel 10j. This change allows the solution to be found almost immediately, as the next choice for vertex (f) is green, which constrains vertex (g) to pink, (j) to blue, (h) to pink, and (i) to red.

While the example above may seem only incrementally better than forward checking, as we will see in Section 8, constraint propagation with MAC adds another large improvement to the backtracking algorithm from forward checking. This is because this algorithm can prune off large sections of the search tree by performing a search of the local neighborhood at each recursive backtracking step.

In our implementation of backtracking with constraint propagation we used the algorithm AC-3 introduced

by Mackworth in 1977 [9]. This algorithm is described in depth by Figure 6.3 of Russel and Norvig [2]. At the heart of AC-3 is a function denoted REVISE. The task of REVISE is to loop through the possible colors i in one vertex and make sure that it is consistent with all the possible colors j in a neighboring vertex. If any i is incompatible with all of j, it is deleted from the list of possible colors, and REVISE returns `true`, indicating that the domian i has been reduced. Otherwise, REVISE returns `false` since i remained unchanged.

AC-3 uses a queue of remaining vertices and the function REVISE to propagate constraints across the whole graph, while simultaneously maintaining arc consistency. For each vertex in the queue AC-3 calls REVISE to see if the vertex provided any new information. If REVISE returns `true`, AC-3 first checks if the domain i is zero, if so, AC-3 returns `false`, signifying an arc-consistency violation. If the domain of i is nonzero, AC-3 propagates the information obtained through REVISE by adding the vertex's neighbors to the queue. AC-3 continues until its queue is empty and returns `true`.

Similarly to backtracking with forward checking, backtracking with constraint propagation makes a small modification to the function BACKTRACK, but instead of calling `forward_check()` before the next recursive call, we will ensure that AC-3 evaluates to `true` before the next recursive call to BACKTRACK.

6.2 Experimental Approach

Again, the experimental approach for this backtracking algorithm does not differ greatly from the previous two versions of backtracking. We do note however, that constraint propagation and MAC do have an increased read an write overhead as compared to forward checking and simple backtracking. Since constraint propagation introduces even more overhead than forward checking on top of backtracking, in Section 8, we will see at what point, if ever, constraint propagation outpaces backtracking.

6.3 Results

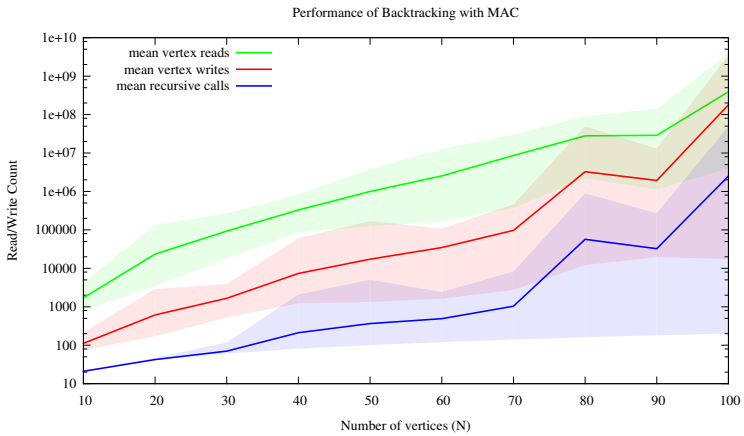


Figure 11: *Logarithmic plot describing the performance of the backtracking with constraint propagation algorithm vs. the number of vertices. The shaded regions represent the minimum and maximum values of each quantity.*

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 some of the classic mechanisms seen in the natural world such as crossover, mutation, and survival of the fittest[10].

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 12 looks like multi-point crossover because the vertices of the graph are unordered when crossover is performed.

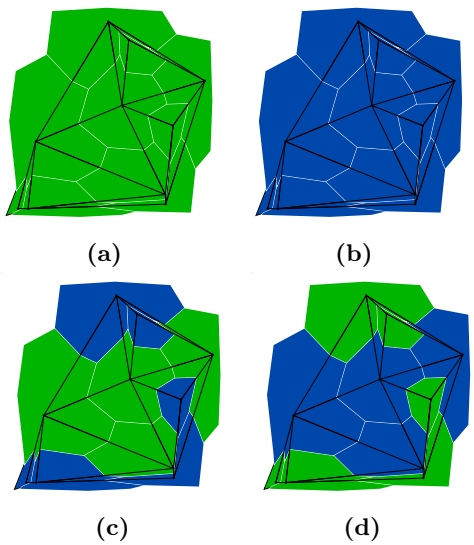


Figure 12: *Simple example of crossover where (a) and (b) are the parents and (c) and (d) are the offspring.*

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 13 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.

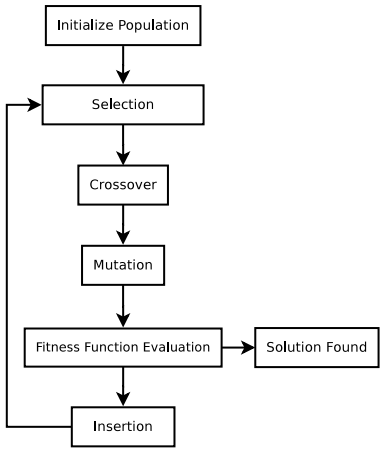


Figure 13: *Flow graph of our genetic algorithm.*

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.

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 vertex. 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 vertex. This metric scaled with the size of the graph for the same reasons as number of reads.

7.3 Results

Figure 14 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 $a < 1$.

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 and therefore $a < b$. Additionally we can see that the slope is less than one so we can further constrict b to $a < b < 1$.

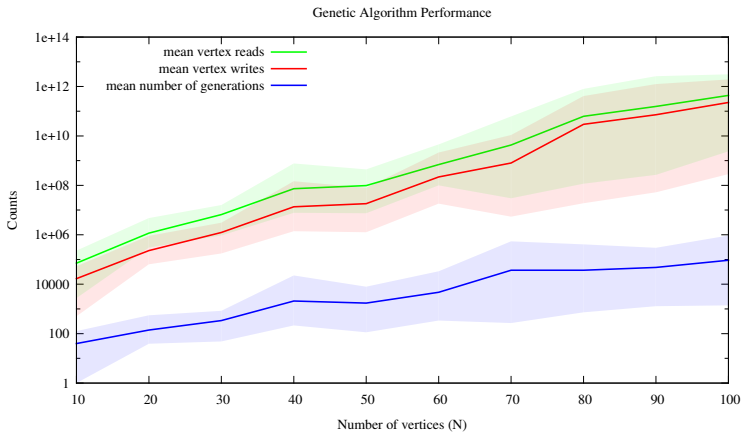


Figure 14: *Logarithmic plot describing the performance of the genetic algorithm vs. the number of vertices. The shaded regions represent the minimum and maximum values of each quantity.*

8 Comparing Algorithm Performance

8.1 Experimental Approach

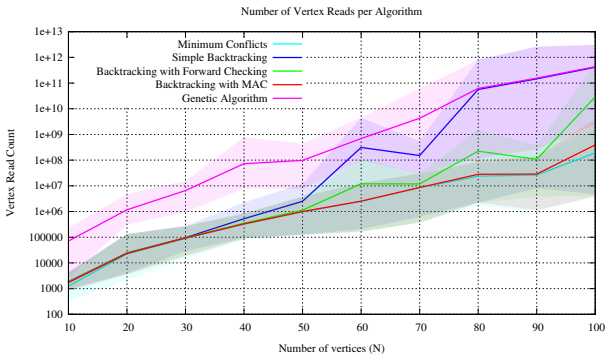
As described in the preceding sections, the number of vertex reads and number of vertex writes were our main metrics for comparing algorithm performance. Under these metrics, all calculations performed by each algorithm were assumed to take the same amount of time between subsequent reads and writes. This assumption is obviously untrue, but tests on our algorithm showed that this assumption was certainly reasonable because the rates (reads/s or writes/s) for each algorithm differed by only a constant factor. This constant factor could be large, but will be dwarfed by the tyranny of the exponential nature of the graph coloring problem.

Even though we are confident that the vertex reads and writes is an accurate metric, to provide a counter-example we also measured the time required by each algorithm to find a solution. This metric is generally considered to be tenuous, as many things can affect a program's execution time, such as the hardware its being run on, what other programs are being run at the same time, etc. We have noted these issues and have made every effort to make this test more accurate. These efforts include running all of the algorithms on the same machine to negate hardware differences, running only one algorithm at a time to eliminate the possibility of a memory bottleneck, and making sure that no other programs are running on the machine at that time. Another contentious facet of execution time metrics, is that one algorithm may simply be more optimized than another, and comparing a poor implementation of min-conflicts against a good implementation of backtracking, for example, would not be a very accurate test. This issue is much harder to mitigate and will not be described in detail here.

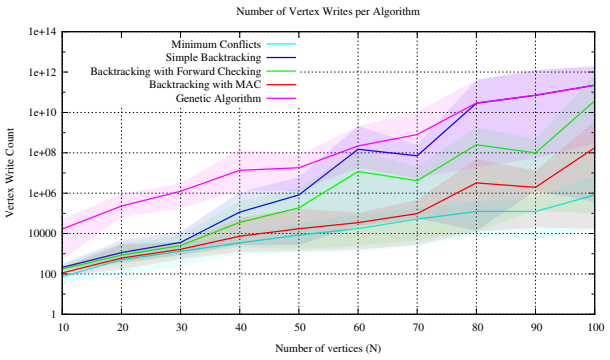
8.2 Results

In general, the minimum conflicts was the fastest algorithm for large N, but it was by no means the clear winner as we expected. In terms of reads, min-conflicts was just as fast as backtracking with MAC up until $N = 100$, where it gained a marginal lead, ending up in first place at an average of about 100 million reads per solution. In terms of writes, it was obvious that minimum conflicts was growing out the slowest out of any algorithm from the start, completing an $N = 100$ graph in over two orders of magnitude fewer writes than the second place algorithm, backtracking with MAC. Surprisingly, min-conflicts was not the fastest algorithm in terms of elapsed time until $N = 100$. We are not sure how to explain this phenomenon, but it is an indication that measuring vertex reads and writes may not be the appropriate measurement to be making.

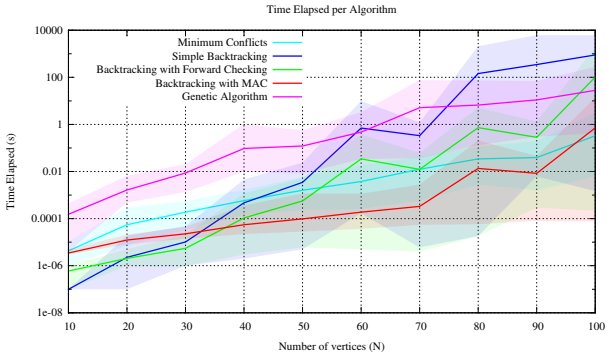
Simple backtracking started off being just as fast as the other two variations of backtrack and min-conflicts,



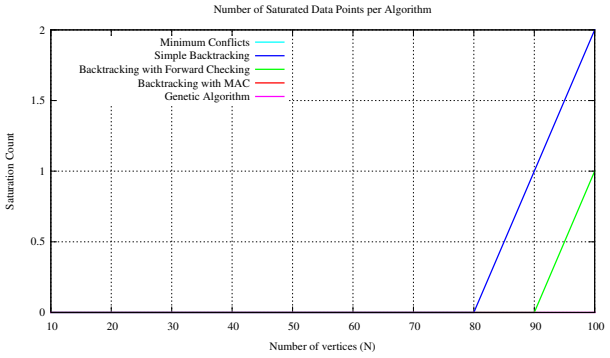
(a) *Average number of vertex reads for each algorithm.*



(b) *Average number of vertex writes for each algorithm.*



(c) *Average amount of time elapsed for each algorithm.*



(d) *Total number of experiments that reached the computation limit.*

Figure 15: *Logarithmic plots comparing the performance of the five different GCP algorithms against four different measurements. Shaded regions represent the maximum and minimum values.*

but grew at a rate faster than any of the other algorithms, finishing in the same place as the genetic algorithm. However, the average number of reads and writes is misleading in this case. Notice how the maximum and minimum values of simple backtracking span the range of almost every other algorithm combined in all three graphs in Figure 15. This indicates that simple backtracking is sometimes just as fast as the min-conflicts and backtracking with MAC, but it can also be much slower, at least five orders of magnitude slower in terms of reads at $N = 100$. Furthermore, we also note that simple backtracking hit the computation limit in 2 out of 20 runs at $N = 100$. This implies that the actual mean would be higher if this limit had not been reached. Interestingly enough, simple backtracking is the fastest out of any algorithm for the $N = 10$ case in terms of elapsed time, indicating that it requires the least overhead of any algorithm, at least in the beginning.

Backtracking with forward checking was a huge improvement on simple backtracking, finishing an order of magnitude faster in reads, writes and elapsed time at $N = 100$. However, forward checking was also slower than backtracking with MAC by more than two orders of magnitude. Forward checking was the fastest algorithm in terms of elapsed time between $N = 20$ and $N = 30$. This region where the additional overhead from constraint propagation is unnecessary, but the raw speed from simple backtracking is not enough to solve the problems quickly.

Almost as fast as min-conflicts, was backtracking with MAC. In terms of reads, min-conflicts and backtracking with MAC seem to growing nearly at the same rate, only diverging at the last point. It would be interesting to see if min-conflicts could retain its lead for larger values of N. In terms of vertex writes, it seems that backtracking with MAC cannot compete with min-conflicts, growing at a much faster rate to finish two orders of magnitude slower than min-conflicts at $N = 100$. But, just like the other backtracking algorithms, backtracking with MAC is initially faster than min-conflicts in terms of elapsed time. Backtracking with

MAC has more overhead than the other two backtracking algorithms, so it only starts to gain an advantage in terms of elapsed time after $N = 40$. It holds onto this lead until $N = 100$, where it is finally beat by min-conflicts.

The genetic algorithm had much more overhead in terms of reads and writes for low values of N under all three metrics. Fortunately, this overhead payed off for larger values of N, as the genetic algorithm grew slower than most of the other algorithms, finishing in the same amount of reads and writes as simple backtracking at $N = 100$. In terms of elapsed time, the genetic algorithm did even better, finishing in the middle of the pack behind min-conflicts and backtracking with MAC. From the trends in all three plots, it seems reasonable to think that the genetic algorithm would eventually become more efficient than all of the backtracking algorithms as N increased beyond 100. It is harder to determine if the genetic algorithm would ever be faster than minimum conflicts. From the read/write plots, it would seem that the time-complexity of the genetic algorithm is growing faster than the time-complexity of min-conflicts. However, from the elapsed-time plot, it seems that the time complexity of the genetic algorithm is growing slower than min-conflicts, so obviously more research would be required.

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] Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1):161 – 205, 1992.

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

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

[7] 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`.

[8] 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`.

[9] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99 – 118, 1977. `http://www.sciencedirect.com/science/article/pii/0004370277900078`.

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