Search, Constraint, and Graph Coloring

- CSCI 446 Artificial Intelligence -

Group 16
James Beck, David Rice, Tyler Wright

Montana State University
Gianforte School of Computing

Abstract

Among NP-hard problems, the graph coloring problem is often used as a base example in heuristic demonstrations on approximating solutions that might otherwise be unsolvable. Given a planar connected where edge intersections are not allowed, the goal is to find colorings for the graph such that no edge connects two of the same colors and the number of colors used is less than some value k. In the case of these experiments, k values of 3 and 4 were used. The goal of this paper was to compare the performance of five different graph coloring algorithms (simple backtracking, backtracking with forward checking, backtracking with constraint propagation(MAC), min-conflicts and a genetic algorithm), and determine which was better suited for the graph coloring task. Two metrics were used to gauge performance, the number of decisions made by each algorithm and the number of successful colorings on graphs of a given size. Though it was originally hypothesized that that the simple backtracking solution would do poorly and be used mostly as a base comparison for the other algorithms, the results showed that the simple backtracking solution, on average, often outperformed all others. However, the backtracking solutions were found to have weaknesses in that they would often not find solutions until very late in the traversal process whereas the the genetic algorithm was much more consistent in amount of decisions taken. This consistency suggests that as graph size grows past the bounds used here (100 nodes), that the genetic algorithm would likely become the most efficient at finding solutions.

Contents

1 Problem Statement

Consider a planar connected graph $G = (V, E)$. The Graph Coloring Problem attempts to assign k-colors to the vertices such that no adjacent vertices have the same color. That is, given valid color set C with $|C| = k$, vertex $v \in V$ with vertex color $c(v)$, and adjacent vertex set E_v, the graph is valid if

$$\forall e_{ij} \in E, \;\; c(v_i) \neq c(v_j) \tag{1}$$

The chromatic number $\chi(G)$ of G is the smallest $|C|$ such that the graph is still valid. Given the graph for this problem is planar, the Four Color Theorem [3] guarantees a valid coloring for $k = 4$.

This paper investigates the search for a valid coloring for G when $k = 3$ and $k = 4$ using five search-based algorithms. Each algorithm takes 10 randomized instances of G for graph sizes $|V| = \{10, 20, 30, ..., 100\}$. Metrics of interest are the number of decisions each algorithm made and whether it found a solution for $k = 3$ and $k = 4$.[1] The five algorithms are:

1. Min-Conflicts
2. Simple Backtracking
3. Backtracking with Forward Checking
4. Backtracking with Constraint Propagation
5. Local Searching Using a Genetic Algorithm with a Penalty Function and Tournament Selection

1.1 Hypothesis

A hypothetical rank from "best" to "worst" would be the backtracking with constraint propagation, genetic algorithm, min-conflicts, backtracking with forward checking, and finally simple backtracking. We have two approaches for how to rank an algorithm's performance:

1. Devise a function $performance = f(d, s)$ where d is the rate of decisions made for each graph size and s is the number of examples solved.
2. Rank each algorithm based off its performance for each individual category $[d, s]$. Discuss which categories are more important in which situations.

[1] Note an optimal solution is not being searched for, rather any solution given the constraints.

Min-Conflicts The min-conflicts makes decisions based solely on conflicts between nodes of a random initial assignment. The advantage is that if the random assignment is close to the solution, the algorithm finds it fairly fast. This can be seen in the following tables when minimum decisions made is low. However, this advantage is also a curse because if the random assignment is far away, min-conflicts slows down. This can be seen when the maximum decisions made is high.

Simple Backtracking As a graph coloring algorithm, true to its name, the Simple Backtracking solution does not rely on many complicated decisions when attempting to find a solution. Despite the fact that each iteration of the algorithm does not contain much code, the number of different node colorings should be quite high. Since there is no validity check before coloring, the number of nodes visited, and therefore decisions made, is expected to be much higher than the two more sophisticated backtracking versions. It is expected that decisions made for Simple Backtracking will be the base level and be greater than for each of the other four algorithms.

Backtracking with Forward Checking The forward checking algorithm is very similar to the simple backtracking solution with the exception that it will not iterate through colors that it knows are invalid with respect to the neighboring nodes. These checks do add a small amount of decisions but overall, having to traverse less nodes is expected to offset this overhead and put the forward checking algorithm ahead of the simple version. However, this small improvement in number of visited nodes is not expected to be enough to put the decisions made metric below any of the other three algorithms.

Backtracking with Constraint Propagation The constraint propagation algorithm is the most complicated and has a much higher ratio of decisions made to iterations but is likely to make up for this in the low number of iterations that should be executed when compared to the other backtracking solutions. With the ability to look much further down a given path, the propagation algorithm should eliminate invalid colorings much earlier. Due to this level of foresight, it is expected that this algorithm will be able to perform better than even the Min-Conflicts solution for smaller graph sizes. On the larger graphs above sixty nodes, the convergence of the min-conflicts and genetic algorithms will likely speed up at a less linear rate than with the constraint propagation and should still outperform the later.

Genetic Algorithm The genetic algorithm is the primary reason we are comparing the rate of average decisions to graph size rather than just average decisions. Because a GA interacts with n graphs at a time rather than just the vertices of one graph, it is highly expected it the will make more decisions with a magnitude of $O(n^a)$ or even $O(a^n)$. However, regarding the rate of decisions to graph size, We hypothesize that the GA will perform only

slightly worse than the rate of other algorithms. That is, for the GA time complexity A and all other algorithm's time complexity B

$$A = O(qn^a), B = O(rn^a) \quad | \quad r \leq q \tag{2}$$

The GA may make up for its worse time complexity and amount of decisions made by being more likely to satisfy the graph coloring constraint.

We hypothesize that the GA will perform worse than all algorithms for small sizes of $|V|$, but better than all algorithms for large $|V|$.[2] This is because of the extra overhead a GA causes with its population of graphs and need to iterate, inspect, and change values in each individual. A simple recursive or greedy algorithm will quickly find and fix errors for small $|V|$, but those search heuristics quickly become ineffective at large $|V|$ due to the growth of decision investment required to repair already visited values. The GA may excel for large $|V|$ due to the randomized approach in tandem with fitness evaluation. There is a high chance for mutation and crossover to reveal potential solutions in $O(c)$ time that would have taking the other algorithms $O(a^n)$ time.

2 Description of Algorithms Implemented

Five algorithms applied to the graph coloring problem are used and compared: Min-Conflicts, Simple Backtracking, Backtracking with Forward Checking, Backtracking with Constraint Propagation, and Local Search Using a Genetic Algorithm with a Penalty Function and Tournament Selection. Each algorithm will highlight the strengths and weakness of various search-based heuristic approaches such as greedy decision making, hill climbing, survival of the fittest, and local search.

2.1 Min-Conflicts

Min-conflicts is essentially a local search heuristic that makes decisions based on the variables that are in conflict. It starts with a complete assignment that can have a large impact on performance. Generally, choosing an initial assignment that is closer to a solution is better, but cannot always be guaranteed to converge [4]. This assignment can be chosen in a number of ways, such as greedy, hillclimbing, simulated annealing, or simply at random [4]. Purely random would not be an effective technique, rather a hybrid approach that does a quick local search and some random decisions [4]. In other words, start with a few iterations of a local search and then possibly fill in the rest randomly so as to not waste time. If too many iterations are used, one might as well use the local search!

Now with an initial random assignment of colors in tow, only the nodes with conflicts are considered each iteration. To start, one of the nodes with conflicts is chosen, but will not consider that node again in the current step i to prevent oscillation between values.

[2] Assuming the tunable parameters are kept the same for all graph sizes

Algorithm 1 Min-Conflicts

```
1: procedure MinConflicts(steps)
2:     assignment = complete assignment of colors using a greedy local search
3:     for i = 0 to steps do
4:         for all nodes with conflicts do
5:             node = node with conflicts
6:             for all colors do
7:                 procedure Conflicts(node.color)
8:             node.color = color which returns lowest value from Conflicts()
9:         if assignment is a solution then
10:            return assignment
11:    return failure
```

Next, all colors for that node are considered and conflicts in each assignment are checked. This is done by utilizing a $Conflicts()$ function which basically counts the number of local conflicts in the current assignment. The idea is to pick a color which minimizes this count.

2.2 Simple Backtracking

Algorithm 2 Simple Backtracking

```
1: procedure SimpleBacktracking(startNode)
2:     AvailableColors ← all possible colors
3:     for all colors ∈ AvailableColors do
4:         startNode ← color
5:         if SatisfiesConstraint(startNode, color) then
6:             for all nodes ∈ AdjacentEdges do
7:                 nextNode ← node
8:                 SimpleBacktracking(nextNode)
```

Simple backtracking is perhaps the most basic of graph traversal algorithms in that it relies on what is essentially a depth first search. After selecting a starting node in the graph and coloring it, if the coloring is valid, the first edge of that node is then moved to. Only after an invalid path is found from the first edge of a node is the second edge traveled. If a node is unable to be successfully colored, it will return to the previous node, re-color that node and then try the same edge traversal approach once more. When a state is reached where all nodes have been validly colored, the algorithm returns true to the node before and these returns cascade back to the starting node[1].

For the implementation of the simple backtracking algorithm, in order to better compare improvements made in the algorithm with forward checking and constraint propa-

gation, no modifications were made to increase efficiency. The simple algorithm iterates through each node, setting it to the first color in the list of colors and then checking if that coloring is valid. If the coloring is invalid, the next color will be chosen. If there are no colorings that satisfy the constraints, the algorithm backtracks to the previous node and sets that node color to the next color in the color list before continuing forward once more.

2.3 Backtracking With Forward Checking

Algorithm 3 Backtracking With Forward Checking

```
1: procedure ForwardChecking(startNode)
2:     AvailableColors ← colors not taken by adjacent nodes
3:     for all colors ∈ AvailableColors do
4:         startNode ← color
5:         if SatisfiesConstraint(node, color) then
6:             for all nodes ∈ AdjacentEdges do
7:                 UpdateConstraints()
8:                 nextNode ← node
9:                 ForwarChecking(nextNode)
```

The forward checking algorithm functions similarly to the simple algorithm with the exception that each node keeps track of the colorings of all nodes adjacent to it and will therefore only iterate through valid colorings. In order to account for changes in node constraints during the backtracking steps, individual constraints are checked prior to each coloring of that node. If no valid colorings are possible, the algorithm will backtrack to the previous node, re-color that node and update the valid colors for the surrounding nodes[1]. The backtracking method is kept the same as that in the simple backtracking algorithm so that any observed improvements can be attributed to the changes in node constraints.

2.4 Backtracking With Constraint Propagation (MAC)

Like the backtracking algorithms using the simple and forward checking approaches, the Constraint propagation algorithm iterates through the graph in a depth first fashion. However, in order to reduce the amount of incorrect paths chosen, the constraint propagation algorithm will use the AC-3 arc constraint algorithm to monitor whether or not all nodes in the graph satisfy the constraints[1].

To accomplish this propagation, each colored node will remove the coloring assigned to it form the uncolored nodes connected to it. Each uncolored node will evaluate the current valid colorings that are available to it and if it only has one available color, will remove that color from the valid colorings of all uncolored, connected nodes. When a node

Algorithm 4 Backtracking With Constraint Propagation (MAC)

```
1: procedure ConstraintProp(startNode)
2:     AvailableColors ← GetAvailableColors(startNode)
3:     for all colors ∈ AvailableColors do
4:         startNode ← color
5:         if SatisfiesConstraint(startNode, color) then
6:             UpdateAllNodeConstraints(startNode, color)
7:             if Conflict not found then
8:                 for all nodes ∈ AdjacentEdges do
9:                     nextNode ← node
10:                    ConstraintProp(nextNode)
```

Algorithm 5 Propagate Valid Colors

```
1: procedure Propagate(startNode)
2:     AvailableColors ← GetAvailableColors(startNode)
3:     for all node ∈ adjacentNodes do
4:         if not node.colored then
5:             if not startNode.colored then
6:                 node.validColors ← node.validColors - startNode.color
7:             else
8:                 node.validColors ← node.validColors - startNode.validColors[0]
9:             if node.validColors is empty then return false
10:            Propagate(node)
```

is found that has an empty list of valid colorings, the propagation algorithm will return false and the last visited colored node will be re-colored.

Like the backtracking algorithm with forward checking, the constraint propagation keeps all aspects except the node coloring choices constant with regards to the simple algorithm solution.

2.5 Local Search Using a Genetic Algorithm with Tournament Selection and a Penalty Function

2.5.1 Overview

A generic genetic algorithm (GA) can solve constraint problems by creating a population of candidate solutions, generating offspring from the population through crossover based on fitness, mutating the offspring, generating the next generation of population individuals, and repeating this process until a solution is found or maximal generations have passed. The unique design of a genetic algorithm takes place in how crossover and mutation is enacted, how fitness is determined, and various extensions and modifications to existing population individuals takes place.

This specific algorithm extends the generic genetic algorithm by choosing which two parents generate a child through tournament selection. A penalty function is also used to kill the worst members and replace them with new individuals before every successive generation. The components of this genetic algorithm are defined as follows:

- Population: A collection of $|P|$ different colorings of graph G.
- Individual: A specific coloring of graph G that is in the population.
- Genome: The collection of vertices $v \in \mathbf{V}$ with colors $c \in \mathbf{C}$.
- Chromosome: The specific color value c of a specific vertex v.
- Chromosome Fitness: Denoted as $f(v)$. This is a boolean value that evaluates to true if the current chromosome is not in conflict with any of its adjacent vertices and false if an adjacent vertex has the same color as v_c.
- Individual Fitness: Denoted as $f(I)$. This is the integer value of how many chromosomes in the genotype evaluate to $f(v) = true$.

Because each individual in the population has the same value $|V|$, this is a maximization problem $max(f(I))$. The graph coloring constraint is satisfied when $f(I) = |V|$.

Algorithm 6 assumes k is static. The evolutionary loop exits once a satisfying individual is found or once a specified maximum evolutions has occurred. The tunable parameters of interest are the mutation rate, penalty size, tournament size, and population size. The values of these parameters are discussed in detail in section 3.

Algorithm Description: This GA initializes the population with randomized chromosome values for each individual and then enters the evolutionary loop. Children individuals are generated by applying crossover off of two parent individuals. The parent individuals are chosen through tournament selection where n individuals are selected at random and the most fit of those chosen is selected as a parent. Crossover is shown in algorithm 8. Take the first half of the genome of parent P_1 and the second half of the genome of parent P_2, combine the two sub-genomes to create a new genome which is applied to child individual C_1. To create C_2, do the same process but switch P_1 and P_2.

The set of children are then mutated by picking random chromosomes in each individual and changing the chromosomal value to a new random color. Mutation is applied at a density rate specified by a parameter and typically ranges from 5% to 20%.

A penalty function is then applied to the population which replaces the least fit n% of individuals and replaces them with new individuals with randomized chromosomes where n is a tunable parameter. This penalty follows the death penalty described by Yeniay [6] in his paper about penalties for constrained optimization problems, but instead of killing unfeasible solutions, the least-fit solutions are removed. A repair type of mutation that follows the ideas presented in Orvosh and Davis's paper on repair functions in genetic algorithms with feasibility constraints [2] is then applied. It targets n% of chromosomes in each individual and changes their color value to be the value which maximizes fitness.

Finally, the population is checked for an individual that satisfies the constraint. If one is not found, the cycle repeats.

Algorithm 6 GA With Penalty Function and Tournament Selection

```
1:  procedure GENETIC-ALGORITHM
2:      P ← collection of individuals with randomized chromosome
3:      repeat
4:          C = ∅ ← collection of child individuals
5:          repeat
6:              Q ⊂ P ← |q| random individuals from P
7:              p_1, p_2 = TOURNAMENT-SELECTION(Q_1)
8:              c_1 = CROSSOVER(p_1, p_2)
9:              c_2 = CROSSOVER(p_2, p_1)
10:             C ∪ {c_1, c_2}
11:         until |C| = max_C_size
12:         for all p_n ∈ P do
13:             MUTATE(p_n)
14:         for r ← the (penalty_rate * |P|) min(f(P)) individuals do
15:             P − r
16:         satisfied ← true if a valid p ∈ P is found
17:     until satisfied ∨ max_generations_reached
```

Algorithm 7 Tournament Selection

procedure TOURNAMENT-SELECTION(Q)
Require: $Q \subset P$
    $p \leftarrow max(Q)$ ▷ max means most fit
return p

Algorithm 8 GA: Crossover

procedure CROSSOVER(p_1, p_2)
Require: $p_1, p_2 \subset P$
    $g \leftarrow \emptyset$
    for first half of $p_1.genome$ **do**
        $g \cup p_1.genome$
    for second half of $p_2.genome$ **do**
        $g \cup p_2.genome$
    return g

2.5.2 Analysis

The GA needs to maintain diversity while providing opportunities for high-potential individuals to survive through generations. Crossover is the primary mechanism used to accomplish this. If crossover causes the children to look too similar to their parents, the population will eventually hit an unsatisfied asymptote where the colors of each vertex in each individual graph are the same. If crossover does not provide a chance for children to find the fit elements of their parents, children will usually be much less fit than their parents as $\lim_{f \to |P|} avg(f(P))$.

A key concern in algorithm 6 is that the single-point crossover does not encourage enough diversity. In fact without the mutation function, the population often ended with $|P|$ instances of near identical graph colorings that never converged. To accommodate this, the child set was set to be larger than the parent set and the mutation rate was set to be quite high at a value of 30%. Furthermore, once all individuals had mutated, a second layer of mutation was applied that changed a low percentage of chromosomes to the color that caused lowest chromosome fitness value for that specific chromosome. Results and convergence frequency improved dramatically once the two layers of mutation were added.

3 Experimental Approach

The experimental approach is three-fold. There needs to be an approach to the code architecture such that each algorithm can be done by an independent person without having to worry about dependencies on other parts of the program. Furthermore, there needs to be

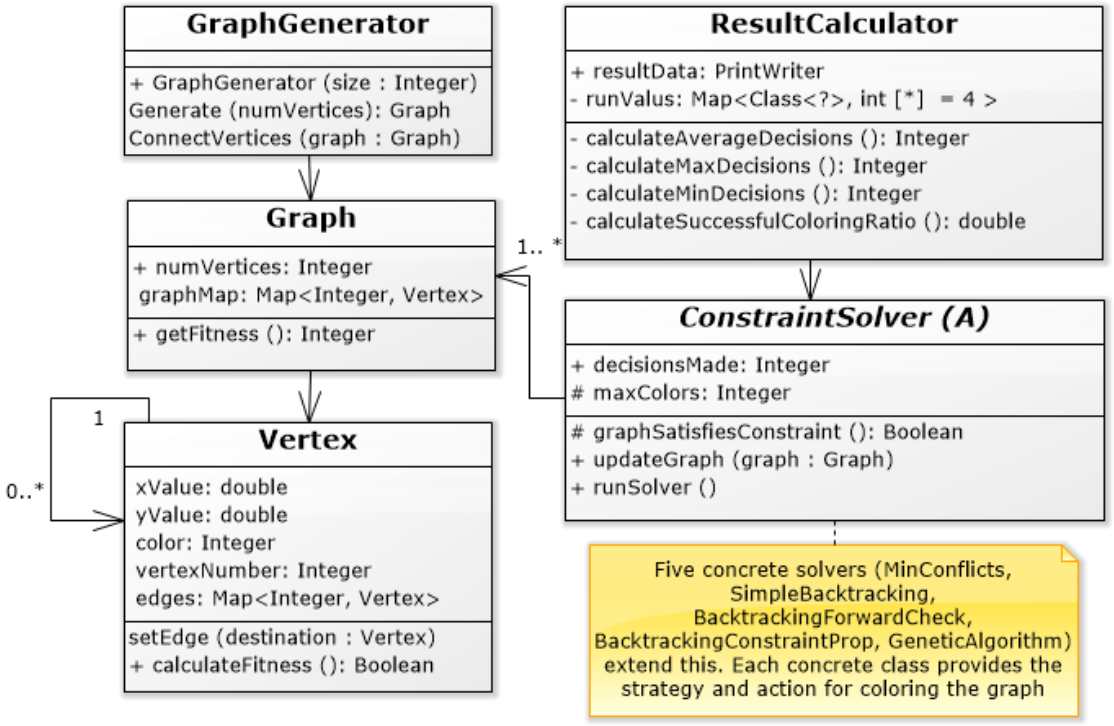
an ability to run a suite of tests in an automated fashion. This necessitates doing multiple runs of many graphs of many sizes on many algorithms. Section 3.1 describes the concrete implementation.

Secondly, the experimental approach needs determinations on which metrics are valuable. The notably different approaches used by each algorithm, restriction of allowed colors, and randomized nature of the graph eliminates many of the more traditional metrics used in graph coloring analysis. Section 3.2 describes the metrics used.

Finally, parameters must be tuned throughout the experimental process. Parameter turning is discussed in section 3.3.

3.1 Program Architecture

Figure 1: Graph Coloring Project Architecture



See figure 1 for the UML representation of this architecture. Each algorithm class will extend the ConstraintSolver class which enforces each algorithm provides the number of decisions made. The abstract ConstraintSolver class also defines a method SatisfiesConstraint. The ResultCalculator is provided *decisionsMade* and *satisfiedConstraint* from each constraint solver immediately after each solver finishes running its procedure on one graph. ResultCalculator stores the data from each run in a map that links

each solver to its data over any amount of runs. Finally, the ResultCalculator runs calculations to provide the following metrics for each solver:

1. Average decisions made.
2. Most decisions made on a single graph.
3. Least decisions made on a single graph.
4. The ratio of graphs correctly colored to graphs attempted.

3.2 Metrics

A typical investigation into graph coloring problems looks to minimize the chromatic number needed to produce a valid coloring of graphs of various densities such as the what Hindi and Yampolskiy [5] utilize in applying a genetic algorithm to the graph coloring problem where:

$$\chi(G) = min(k : P(G,k) > 0) \tag{3}$$

for color k, and number of solutions for a graph with k colors $P(G,k)$.

However, this paper looks to compare the performance of the five algorithms described in section 2. We use the number of decisions made and percentage of graphs successfully colored for a static $k = 4$ and $k = 3$ as metrics.

Decisions Made: The genetic algorithm will have vastly more decisions made than the other algorithms due to its application of decisions onto n concurrent graphs over many generations. Thus the metric of interest is not strictly decisions made, but the rate of decision growth as the graph size grows.[3] In other words, for $x = graph\_size \in 10, 20, ..., 100$ and $y = decisions\_made$ the decisions metric of one algorithm d is,

$$d = O(f(x,y)) \tag{4}$$

Successful Colorings: The other metric is more easily comparable. It simply compares the effectiveness of each algorithm based on the percentage of successful colorings for each graph size. Even though there will always be a solution when $k = 4$ for any graph size, many of the algorithms do not converge depending on the graph's construction.

3.3 Parameter Tuning

3.3.1 Genetic Algorithm

Child Mutation Rate The child mutation rate affects the percentage of nodes that are not already a valid coloring in each individual that has its color changed to another random

[3]Cases where a valid solution was not found are not included in the average decisions made.

color. This was set to a high value 20% because the crossover strategy did not introduce diversity in effective ways.

Repair Mutation Rate At the end of each generation, a percentage of chromosomes in each individual are fixed to $max(f(c))$ where c is the current chromosome. This mutation happens immediately after the random mutation and targets 15% of the population due to help stem the ineffective randomizations of the chromosomes from the previous mutation.

Penalty Size This is the amount of least fit individuals to cull from the population and replace with randomized new individuals. Because so much randomization is already happening in the mutation phase, this is set to remove only the single least fit member.

Tournament Size This value was set to 2 in order to help avoid over-fitting by lowering the chance the the most fit individuals were chosen too often to reproduce.

Population Size The tests were run on a population size of 30. The GA was already running remarkably slow, and $|P|$ reliably converged on graph sizes up to 60.

3.4 Experiment Flow

Average values are needed for each algorithm run on a specific graph size due to the natural variance the randomized graphs will cause. Thus, given $G = (V, E)$ for each graph size $|V|$, 20 randomized instances will be run against each algorithm with the average, max, and min values over the twenty runs sent as output. This is shown in algorithm 9

Algorithm 9 Experiment Flow

```
1:  procedure RUN-EXPERIMENT
2:      num_runs = 10                          ▷ 10 different graph sizes
3:      max_colors = 3, 4
4:      graph_size = 10                        ▷ First suite will be on graph size 10
5:      suite_iterations = 20                  ▷ 20 randomized graph for each graph size
6:      while run ≤ num_runs do
7:          while suite ≤ suite_iterations do
8:              graph ← NEW-GRAPH(graph_size)
9:              for all solver ∈ Algorithms do
10:                 GRAPH-COLORING(solver, max_colors)
11:             suite + 1
12:         graph_size + 10
13:         run + 1
```

4 Experiment Results

For all numerical results except for the ratio of successful colorings, data is only from cases when the algorithm found a solution. The abbreviations are:

1. MC: Min-Conflicts
2. SB: Simple Backtracking
3. BFC : Backtracking with Forward Checking
4. BCP : Backtracking with Constraint Propagation
5. GA : Genetic Algorithm

4.1 Four Coloring

The algorithms satisfied the constraint for $k = 4$ quickly and effectively for smaller graphs. Once $|V| \geq 50$, MC failed to find valid colorings, and the BFC started making exponentially more decisions. The backtracking algorithms did not always have a 100% success rate. This is notable given that the backtracking algorithm should always find a solution [4] Details of this are discussed in section 5.

4.1.1 Four Coloring Satisfaction-Ratio and Decision Values

Algorithm	*Avg. Decisions*	*Max Decisions*	*Min Decisions*	*Satisfied Ratio*
		Graph size 10		
MC	1,989	8,496	77	1.000
SB	73	145	59	1.000
BFC	248	415	218	1.000
BCP	350	669	414	0.800
GA	909	1,767	609	1.000
		Graph size 20		
MC	19,674	113,214	567	1.000
SB	301	1,666	121	1.000
BFC	1,549	6,795	838	1.000
BCP	354	1,481	1,441	0.250
GA	11,924	38,661	4,062	1.000

[4] Even at worse case, it will have changed all colors on all nodes until the constraint was satisfied.

Algorithm	*Avg. Decisions*	*Max Decisions*	*Min Decisions*	*Satisfied Ratio*
		Graph size 30		
MC	190,014	768,488	9,855	0.950
SB	12,271	23,5588	187	1.000
BFC	8,8013	1,683,624	1,858	1.000
BCP	796	3,461	3,094	0.25
GA	91,857	401,948	12,882	1.000
		Graph size 40		
MC	330,420	988,339	2,538	0.750
SB	3,568	27,673	317	0.850
BFC	26,853	188,122	3,792	0.850
BCP		Did Not Solve		
GA	676,264	7,561,558	31,319	1.000
		Graph size 50		
MC	63,884.60	1,277,692	1,277,692	0.050
SB	31,755	477,417	328	0.950
BFC	321,947	5,071,172	5,208	0.950
BCP		Did Not Solve		
GA	2,564,917	22,043,538	71,232	0.950
		Graph size 60		
MC		Did Not Solve		
SB	136,544	2,198,044	428	1.000
BFC	1,902,743	31,854,502	7,996	1.000
BCP	648	12,964	12,964	0.050
GA	1,712,152	6,796,364	147,227	0.900
		Graph size 70		
MC		Did Not Solve		
SB	56,496	760,038	720	0.800
BFC	702,684	9,572,664	12,757	0.800
BCP		Did Not Solve		
GA	3,510,198	10,080,801	548,937	0.950
		Graph size 80		
MC		Did Not Solve		
SB	3,137,781	60,705,715	579	0.950
BFC	50,391,248	975,570,580	14,324	0.950
BCP		Did Not Solve		
GA	4,588,425	16,570,024	844,023	0.650

Algorithm	*Avg. Decisions*	*Max Decisions*	*Min Decisions*	*Satisfied Ratio*
		Graph size 90		
MC		Did Not Solve		
SB	8,626,512	72,461,884	774	0.850
BFC	54,884,585	1,499,734,884	19,594	0.850
BCP		Did Not Solve		
GA	4,634,637	23,395,796	1,480,007	0.570
		Graph size 100		
MC		Did Not Solve		
SB	3,807,202	43,061,804	3,240	0.900
BFC	72,250,484	729,952,945	52,459	0.900
BCP		Did Not Solve		
GA	8,181,442	25,029,480	1,330,902	0.600

On average, the simple backtracking solution performed the best and consistently found colorings at a rate at or above the other algorithms while doing so in less decisions. However, the simple solution also had a much larger difference between maximum and minimum decisions when compared to the genetic algorithm, suggesting that the genetic algorithm performs much more consistently and is perhaps less affected by graph structure.

Though the backtracking algorithm with constraint propagation was much less successful at finding correct colorings than the other four algorithms, when it did find a solution, it did so in far fewer decisions. There is a possibility that the lower decisions mean that the algorithm has the potential to perform well if modified but it is more likely that the algorithm only found solutions when the would occur in an early path through the graph and therefore if the consistency was improved, these averages would increase greatly.

4.1.2 Four Coloring Time Complexity

The average decisions for each graph is plotted a their best fit quadratic regression curve shown in figure 2. Reference appendix A for the regression equations goodness measures.

Only the SB, MC, and GA are plotted because the MC and BCP algorithms failed to find valid colorings after the graph size grew beyond 50 nodes

4.2 Three Coloring

All algorithms rarely found a solution when only three colors were allowed and $|V| = 10$. They all failed to find a 3-coloring once $|V| \geq 20$. This may be because nearly all graphs were simply not three colorable. That is,

$$\forall g \in G, \chi(g) > 4 \mid |g_v| \geq 20 \tag{5}$$

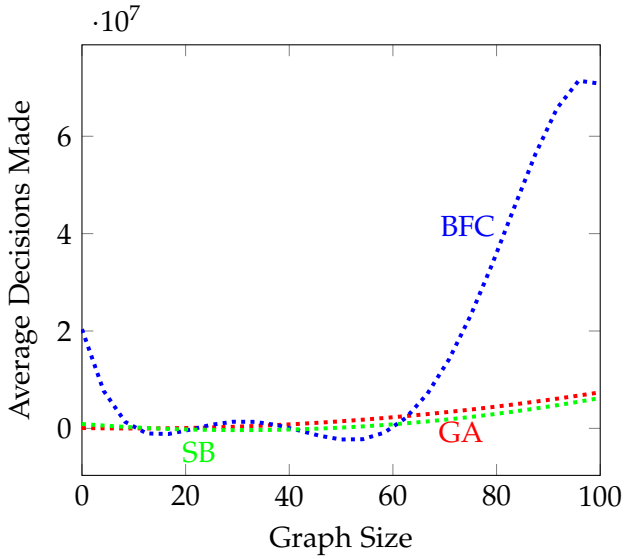


Figure 2: Time complexity (logarithmic scale)

Alternately, it is possible solutions some times existed but the algorithms were not robust and tuned enough to find the correct three coloring. While the task of determining if a graph is 3-colorable is an NP-Complete problem out of the scope of this paper, we make the assumption that most graphs did not have a possible 3-coloring solution.

Given all algorithms never found a solution for $|V| \geq 20$, data is only shown for $|V| = 10$ when $k = 3$.

Algorithm	*Avg. Decisions*	*Max Decisions*	*Min Decisions*	*Satisfied Ratio*
		Graph size 10		
MC	18	360	360	0.050
SB	2.95	59	59	0.050
BFC	10.90	218	218	0.050
BCP	20.75	415	415	0.050
GA	215.30	2458	1848	0.100

Though 3 coloring solutions were only found for graphs of size 10, the fact that this was universal across all algorithms suggests that the lack of valid colorings was most likely do to the fact that larger graphs were not colorable using only 3 colors. Because the backtracking solutions were expected to perform at their best on the smallest graphs, it is no surprise that the simple backtracking solution, with the low amount of logical checks, performed best.

5 Algorithm Behavior

5.1 Min-Conflicts

The min-conflicts algorith showed initial promise on the smaller graph sizes with max colors set at 4 but after size 40, rarely found a solution. The relatively random nature of the color change algorithm meant that a graph could get caught in a state where in would repeat incorrect solutions or simply fall into a cycle of not being able to find a better coloring for any of the nodes while still not satisfying all constraints.

On coloring problems with max colors set to 3, the algorithm did do similarly to the other algorithms in that it only found a solution for graph sizes of 10. Like the 4 coloring problems, during these smaller 3 coloring problems, the min-conflicts algorithm performed relatively competitively.

5.2 Simple Backtracking

Despite all initial assumptions, the simple backtracking solution turned out to outperform the other backtracking algorithms. Though surprising, the performance was likely due to the simplicity and low number of logical operations within the primary methods of the algorithm. Despite the higher performance values, the simple algorithm did have some weak points, particularly when the graph sizes got to be above size 50. In some cases, the valid solution would not be found until toward the end of graph traversal process and due to the unintelligent nature of the algorithm, each coloring attempt would exponentially increase the amount of decisions made. As expected, when the graph sizes were lower, the simple backtracking algorithm was able to consistently find a four-color solution with minimal backtracking.

Though three-color solutions were rarely found for graph sizes greater than 10, this is a result of very few of the generated graphs actually being colorable using 3 colors. There were no instances where a graph was found to be colored using 3 colors in one algorithm and then not in another.

5.3 Backtracking with Forward Checking

While based on the simple backtracking algorithm, only with slightly updated coloring logic, the forward checking solution was expected to outperform the simple due to the high expected cost of attempting a coloring that would immediately be shown to be invalid. After running all algorithms, just as they were in the simple implementation, almost all hypotheses turned out to be false. The only explanation for the higher counts of decisions made is the logic that is used to keep track of which colors were valid for a given node. The small increase in computation, when multiplied thousands of times, must have accumulated enough to offset the benefits of generally having less iterations when compared to the simple algorithm. Besides these differences, the simple and forward checking

back propagation algorithms behaved identically over all graph runs.

5.4 Backtracking with Constraint Propagation

After examining the behavior of the previous two backtracking algorithms, it was less surprising to find that again, the hypotheses for the final backtracking algorithm, with respect to the others, were also wrong. The constraint propagation solution was not only not the most efficient solution, it turned out to perform worse than the two others. Unlike the others, there are possible reasons for this poor performance that may be explained by implementation choices. The most likely reason for the increase in decisions as well as a decrease in solutions found is from a feature that was added to eliminate possible unwanted looping.

With the way that the propagation function was designed, there was the possibility for two nodes to continually attempt to update the other list of valid colorings if they were both uncolored and each had only one possible valid color that also differed from the other valid color. Because of this loop, an update would not be propagated in these cases. While this may seem like a rare occurrence, it is likely that having this inconsistent propagation flow disrupted the paths through the graph and caused some possible solutions to be skipped.

5.5 Genetic Algorithm with a Repair Function and Tournament Selection

The GA exceeded expectations given in the hypothesis by maintaining good performance measure for both small and large graph size values. It was not expected for the GA to fit a polynomial curve with a R^2 value of 0.9044 as seen in figure 2 and appendix A. This polynomial curve may be due to the GA's search space not being heavily affected by $avg(|E|)$ nor the total nodes in the graph given its heuristic of candidate solution search and modification relies on comparison between individuals rather than traversed states in a single graph.

Despite the decent long-term algorithm behavior, the GA consistently took the longest to run often taking tens of thousands of generations to find a best fit solution. The 95% satisfaction ratio for graph size 70 was above expectations, but the GA quickly lost reliability at finding a solution for sizes 80 and beyond. This is likely due to the crossover function. A simple single point crossover does not encourage diversity nor does it guarantee the passing of optimal chromosomes over sub-optimal ones. The application of repair-based mutation also furthers this lack of diversity as similar parent-child individuals will repair the same chromosomes to the same values.

Future Work Should further work happen on this project, I would start modifying the crossover, penalty, and mutate functions in the following ways:

1. Focus more on local search by having the parents copy their most fit neighbor vertex colors to the neighborhood of a target child chromosome at a parameterized rate.

2. Modify the penalty function to not just replace the least-fit individuals, but also search out individuals that are too similar to each other as a function of similarity and fitness.

3. Apply different crossover, mutation, and penalty heuristics depending on the fitness of the individual.

6 Summary

The simple backtracking algorithm seems to perform surprisingly well on average though as graph sizes increase above 40, the time taken tends to vary drastically and often takes a very long time. If a consistent solution is desired, the backtracking algorithms may not be viable, especially as graph size increases. When the forward checking feature was added to the backtracking approach, despite initial assumptions, the algorithm was in fact slowed down without any noticeable improvement in solutions found. Even when the graph size grew and the forward checking was expected to become more beneficial, none of the measured metrics for forward checking ever improved beyond that of the simple implementation. The final solution implementing a backtracking approach, constraint propagation, aldo failed to improve upon the basic algorithm. Though some runs resulted in the constraint propagation algorithm having the lowest measured decisions, it would also rarely find solutions to all the graphs that the other backtracking algorithms solved.

The fourth algorithm implemented, min-conflicts was reasonably competitive with the others on smaller graphs size 40 and under but became very inconsistent when the graph was increased above this level. In many cases, no valid coloring solutions were found while the other algorithms were still finding solutions 90 to 100 percent of the time.

Finally, the genetic algorithm proved to be fairly consistent across all sizes with only a few exceptions. Though it was consistently one of the implementations with the highest number of decisions, it was also fairly predictable in this regard. Unlike the backtracking algorithms that would complete various graphs of the same size in vastly different times, the genetic algorithm would perform similarly on all graph of the same size. The consistency combined with the relatively linear increase in decisions to graph size means that the genetic algorithm may become a better solution as graph size is increased even more. Further work is needed to determine the validity of the genetic algorithm on larger graphs.

References

[1] Roman Bartak. *Constraint Propagation and Backtracking Based Search*. Charles University, Faculty of Mathematics and Physics. URL: http://www.math.unipd.it/~frossi/cp-school/CPschool05notes.pdf (visited on 09/15/2016).

[2] David Orvosh. Lawrence David. *Using a Genetic Algorithm to Optimize Problems with Feasibility Constraints*. 1994. URL: http://dlia.ir/Scientific/IEEE/iel2/1125/8059/00350001.pdf (visited on 09/13/2016).

[3] Georges Gonthier. *Formal Proof – The Four-Color Theorem*. 2008. URL: http://www.ams.org/notices/200811/tx081101382p.pdf (visited on 09/25/2016).

[4] "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems". In: *Artificial Intelligence* 58 (1992), pp. 161–205. URL: http://www.sciencedirect.com.proxybz.lib.montana.edu/science/article/pii/000437029290007K (visited on 09/25/2016).

[5] Musa Hindi. Roman Yampolskiy. *Genetic Algorithm Applied to the Graph Coloring Problem*. 2011. URL: http://ceur-ws.org/Vol-841/submission_10.pdf (visited on 09/12/2016).

[6] Ozgur Yeniay. *Penalty Function Methods for Constrained Optimization with Genetic Algorithms*. 2005. URL: http://www.mcajournal.org/volume10/vol10no1p45.pdf (visited on 09/12/2016).

A Quadratic Regressions of Average Decisions Made

Algorithm	*Quadratic Regression*
MC	not enough successful colorings
SB	$y = 904445 - 83397.79 * x + 1367.191 * x^2$
BFC	$y = 20376380 - 3991978 * x + 258382.1 * x^2 - 7085.756 * x^3 + 83.82818 * x^4 - 0.3431269 * x^5$
BCP	not enough successful colorings
GA	$y = 63357.33 - 17669.34 * x + 910.5789 * x^2$

Goodness Measures		
Algorithm	R^2	aR^2
MC	n/a	n/a
SB	0.6357	0.5316
BFC	0.9443	0.8747
BCP	n/a	n/a
GA	0.9411	0.9243