Solving the Graph Coloring Problem using Genetic Programming

Justine W. Shen
Stanford University
Stanford, California
justine_shen@hotmail.com

ABSTRACT

This paper introduces a method for finding effective graph coloring schemes by deploying genetic programming. The method described here uses automatically defined functions that are helpful in forming highly fit subroutines. ECJ was chosen as the software used in the experiments. All test cases are obtained from DIMACS Challenge benchmark graphs.

1. Introduction and Overview

The problem of graph coloring can be simply stated as the problem of finding an assignment of colors to the vertices of a graph so that adjacent vertices are assigned different colors. The goal is to minimize the total number of colors used in the assignment.

Besides its theoretical significance as an NP-hard problem, graph coloring arises naturally in a variety of applications such as register allocation and timetable examination scheduling. In many applications that can be formulated as graph coloring problems, it suffices to find an approximately optimum graph coloring of the graph with a small though non-optimum number of colors.

This paper presents the technique of finding graph coloring algorithms through the application of genetic programming. The observed outcome shows improved individuals emerging through generations. The best individuals can perform the known optimal coloring for a subset of graphs in the test case set.

The paper first gives some background information in section 2. It then introduces an initial method used in the graph coloring problem in section 3. Section 4 discusses an improved method based on the lessons learned in the initial method. Section 5 states the results and Section 6 provides the analysis. Lastly, section 7 states the conclusion and section 8 provides thoughts on future work.

2. Background

While the graph coloring problem may not look difficult at first glance, its complexity is in fact NP-Hard, which means that finding the optimal solution requires exponential time, based on current knowledge. Therefore, it's not always feasible to find the optimal coloring for graphs of large size. To see this infeasibility, let us assume,

for simplicity reasons, that the size of a graph is defined as the number of nodes (n) in the graph, and that the optimal algorithm takes time 2^n. Solving a graph of 100 nodes may take up to 2^{100} calculations. Even if 1 billion-billion ($1*10^{18}$) operations can be carried out each second, it would still take over 40,000 years to complete the computation needed for finding the optimum solution.

Much research has been done to find fast graph coloring algorithms, often by taking advantage of special properties exhibited by a subset of graphs or by relaxing coloring constraints in near-optimal algorithms. In this paper, we introduce an approach that utilizes genetic programming.

3. Initial Method

This section describes the initial approach used in the graph coloring problem using GP. Reasons for making certain decisions in this approach are presented, and problems encountered are described and discussed. The analysis prompted a change to the initial method. The improved method is presented in section 4.

3.1 Motivation

The problem of graph coloring can be naturally broken down into two subcomponents:

1) The graph needs to be colored correctly. By "correctly", we mean that no adjacent nodes in the graph are assigned the same color.
2) The total number of colors is minimized.

The first condition ensures correctness of the algorithm and the second ensures optimality. Through genetic programming, we hope to generate fit individuals representing good algorithms, covering both conditions.

3.2 Execution and Problem Encountered

A set of functions and terminals is determined such that an individual has the freedom to color a given graph using as many colors as it wants. Actual programming was done in ECJ, and the fitness value was defined as a combination of 1) number of nodes in the graph that are incorrectly colored, and 2) total number of distinct colors used after algorithm completes. Since low fitness number in ECJ actually means good fitness, the fitness definition here is aimed at getting correctly colored graphs as well as bringing down the total number of colors used.

This approach turned out to be surprisingly unsuccessful. As generations develop, the best individual gradually degenerates into one of two extreme cases: the individual either colors the entire graph in one color, therefore producing incorrectly colored graphs, or it uses a different color for nearly each node, therefore producing correct graphs but with virtually no optimization. Because these extreme behaviors were observed, the relative weights of the two components of the fitness value were examined in detail and different weight pairs were tested in an effort to find the perfect balance for both components of the fitness value to go down simultaneously. However, varying the weights did not seem to produce an improvement in

the results.

3.3 Analysis

Two hypotheses were made regarding the causes of the observed failures. The first was that the initial population may not be diverse enough to form a good basis for evolution. The population size used was 1024, and the set of function and terminal nodes do allow formations of reasonably diverse individuals. It's not clear how the population, function and terminal set should be extended to improve evolution behavior. Second, it's possible that the GP algorithm implemented by ECJ does not evolve efficiently when the fitness is based on multiple sub-goals. The evolution process seems to be trapped in local optimums when individuals become specialized in one area and degenerated in the other. This phenomenon is especially undesirable for problems whose fitness values require distinctions among the various sub-goals, and a decrease in the total fitness value does not necessarily guarantee a better individual.

3.4 Modification

Instead of having two sub-goals, the fitness value will represent one goal. This way, any improvement in the fitness value directly translates into a better individual. The other sub-goal of the original fitness value is still necessary, but instead of being a goal, it is transformed into a constraint of the problem. By doing so, we are still looking at the same problem but through a slightly different angle.

4. Final Method

The initial method allows the generated individuals to use as many colors as needed when coloring a given graph. This flexibility forces the fitness value to include a sub-goal, which is to minimize the total number of colors used by an individual algorithm. The trick is to remove this sub-goal from the fitness value calculation and instead put some limit on the number of colors allowed in coloring a graph.

All of the graphs used in the project come from the DIMACS Challenge benchmark graphs. Each graph has a published best-known optimal coloring. Therefore, the final method limits an individual to use no more than the best-known number for coloring the associated graph. The individual either colors the graph correctly using the best-known number, in which case there is a hit, or it will produce some incorrectly colored edges, which is captured by the fitness value. This setup asks GP to only look at the number of edges that are incorrectly colored. When the number goes to zero, we know we have generated an individual that is at least as good as the currently best-known algorithms. Note that the best-known numbers associated with graphs are not necessarily obtained from a single algorithm. Thus, the significance of one individual correctly coloring all graphs bounded by the best-known numbers is high.

4.1 Terminal and Function Set

In the first two major preparatory steps of genetic programming, terminal set and function set are defined for

the graph coloring problem. They are summarized in Table 1 and Table 2 below.

Table 1 Terminal set

Terminal Name	Terminal Description
ColorRand	Assigns a color to the current node by randomly picking a color from the "color set". A color set is a pre-defined set of colors. Each graph has one color set. The number of colors in the color set is equal to the best number known for coloring the associated graph.
ColorSafe	Assigns to the current node a color that is used by a neighbor's neighbor but not by a direct neighbor. If all neighbors' neighbors are also direct neighbors, as in the case of a clique, then no assignment happens for the current node.
Decolor	Changes the current node to be un-colored.
Equalize	Makes all neighbors of the current node the same color.
MakeUnique	Removes the color from all neighbors that have the same color as does the current node.

Table 2 Function set

Function Name	Function Description
IfColored	If current node is colored, run child[0], otherwise run child[1].
IfNotUnique	If current node is not uniquely colored from its neighbors, run child[0], otherwise run child[1].
Progn2	Run child[0] and [child[1] sequentially.
Progn3	Run child[0], [child[1], and child[2] sequentially.

4.2 Sample Individuals

With the defined terminal and function set, an individual generated by GP looks like:

```
Tree 0:
  (IfNotUnique
     (progn2 ADF1[1] ADF1[1])
     ADF1[1])
Tree 1:
  (IfColored
     (IfNotUnique MakeUnique MakeUnique)
     (IfNotUnique ColorRand ColorRand))
```

The above is a best-of-run individual after 5 generations. Whenever this individual processes an uncolored node, it randomly assigns a color to the node. When a colored node is encountered, it will remove the color from all neighboring nodes that have the same color. This method is partially greedy in that it attempts to color every node as they are presented, and assumes that they would be unique locally, and on the other hand does local improvements by removing the color of same-color neighbors.

It fails to generate very good results, because that for the denser graphs, when an uncolored node is assigned the same color as its neighboring nodes, the next iteration would cause all those neighbors to lose their colors. This will take many loops to correct, while ColorRand may introduce more instability.

Figure 1 shows a graphical representation of the same individual.

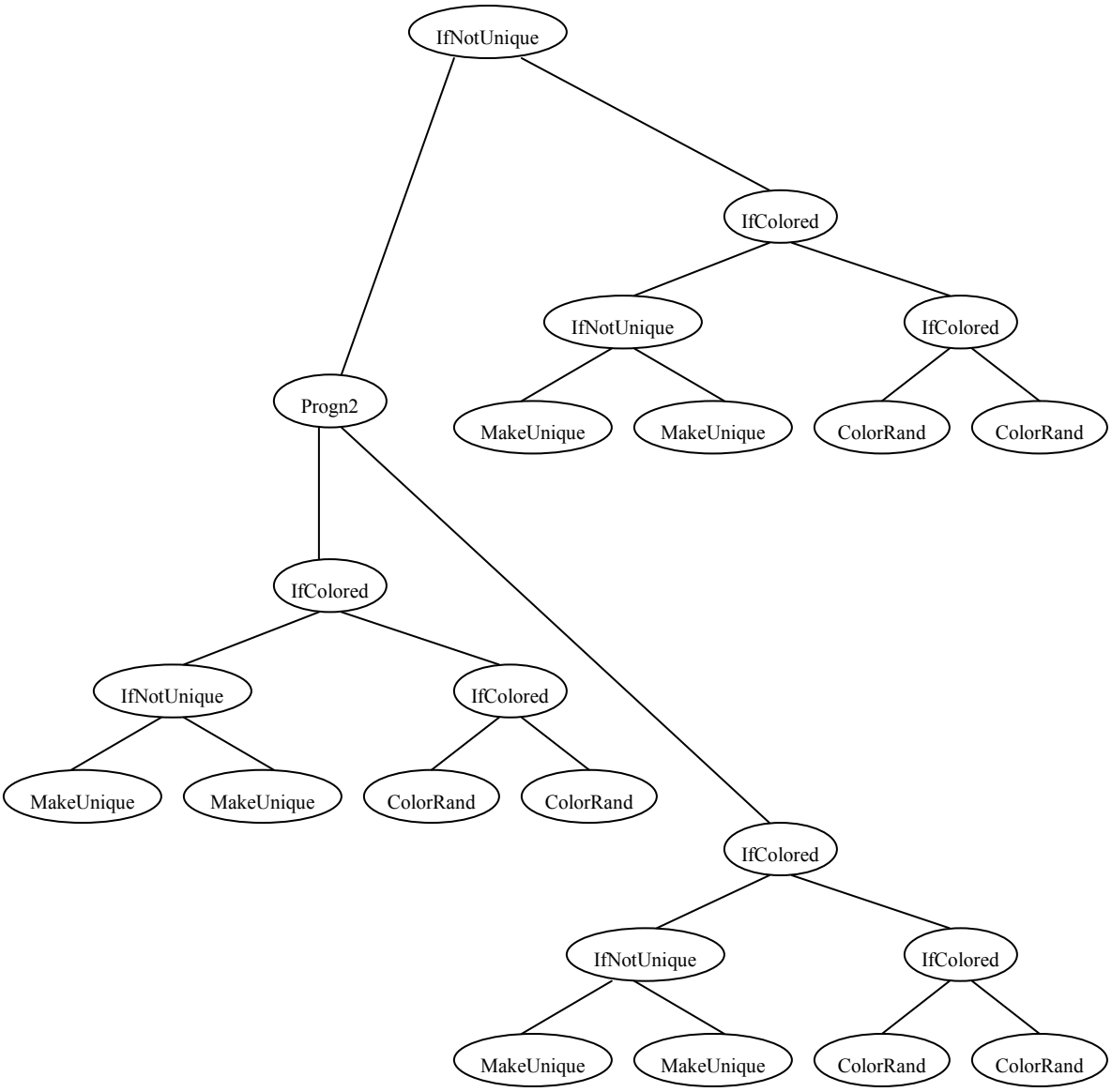

Figure 1 The best-of-run individual after 5 generation

Next is a more complex individual:

```
Tree 0:
  (IfColored
     (IfNotUnique
        (progn2 ADF[1] ADF[1])
        (IfColored ADF[1] ADF[1]))
     (progn2 ADF[1] ADF[1]))
Tree 1:
  (IfColored
     (IfNotUnique
        (IfNotUnique
           (progn2 MakeUnique Decolor)
           Equalize)
        (IfNotUnique MakeUnique MakeUnique))
     (IfColored
        (IfNotUnique Equalize ColorRand)
        (IfColored ColorSafe ColorSafe)))
```

When this individual encounters an uncolored node, ColorSafe is called - this causes the node to be colored with a safe color (one that is unique with respect to its direct neighbors) if possible, while trying to reuse existing colors. If a colored node is not properly colored, then it and all its neighbors with the same color will be de-colored. When a uniquely colored node is encountered, it is left alone. This closely resembles the TABU algorithm, where nodes of a graph are assigned colors that may not represent a legal coloring, and nodes are then re-colored so that a legal coloring may result. The difference is that instead of having a complex step of "changing a node to a different color if it resolves an existing conflict", the GP algorithm introduces an "uncolored state" as an intermediate step before a decision is made about the node's color.

4.3 Test Cases

DIMACS (Discrete Mathematics and Theoretical Computer Science) is founded as national science foundation. DIMACS graph coloring instances are benchmark graphs that cover a variety of graph types and are used in many graph coloring studies. 10 graphs were taken from the DIMACS benchmark graphs as the test cases for this project. They are listed in the following table.

Table 4 Test cases

Name	Nodes	Edges	Optimal Coloring	Description
David.col	87	406	11	http://mat.gsia.cmu.edu/COLOR/instances/david.col
Huck.col	74	301	11	http://mat.gsia.cmu.edu/COLOR/instances/huck.col
Jean.col	80	254	10	http://mat.gsia.cmu.edu/COLOR/instances/jean.col
Myciel3.col	11	20	4	http://mat.gsia.cmu.edu/COLOR/instances/myciel3.col
Myciel4.col	23	71	5	http://mat.gsia.cmu.edu/COLOR/instances/myciel4.col
Myciel5.col	47	236	6	http://mat.gsia.cmu.edu/COLOR/instances/myciel4.col
Queen5_5.col	25	160	5	http://mat.gsia.cmu.edu/COLOR/instances/queen5_5.col
Queen6_6.col	36	290	7	http://mat.gsia.cmu.edu/COLOR/instances/queen6_6.col
Queen7_7.col	49	476	7	http://mat.gsia.cmu.edu/COLOR/instances/queen7_7.col
Queen6_6.col	64	728	9	http://mat.gsia.cmu.edu/COLOR/instances/queen8_8.col

4.4 Fitness Measure

The raw fitness of this problem is the sum, taken over the 10 fitness cases, of the number of incorrectly colored edges in each graph. Another way of measuring fitness can be calculating the total over the number of incorrectly colored nodes. However, calculating over edges gives more weight to incorrect nodes that are connected to more edges, and properly coloring such nodes has more positive impact on the overall correctness of the graph. Some test cases have standalone nodes that are not connected to any edge. In those cases, there won't be edges that account for the correctness of the nodes in the fitness calculation. However, the standalone nodes can be trivially handled by the fact that any color assignment is a correct assignment.

The fitness value can easily be bounded by 0 and 1. Because the number of incorrect edges is bounded by zero and the total number of edges in the graphs, dividing the number of incorrect edges over total edges gives a ratio between 0 and 1. However, this range requirement is not necessary in ECJ. In ECJ's KozaFitness class, standardized fitness and raw fitness are considered the same. Standardized fitness ranges from zero inclusive (the best) to positive infinity exclusive (the worst).

4.5 Control Parameters and Termination Criteria

Various population size and max generation number were used in multiple runs of the problem in order to avoid starvation in the initial individual basis and in the generations. The minimum tree depth for the main tree is 1 and maximum depth is 4, while the minimum tree depth for the ADF is 2 and maximum 6. The crossover rate is 0.8 and the maximum crossover depth is 5. The mutation rate is 0.1.

The search space is bounded by maximum tree depth and the sizes of the function and terminal set. With the main tree max depth being 4 and ADF max depth 6, the entire tree has max depth 9. Each node can have at most 3 children. These children can be any of the 9 functions defined in the function and terminal set. Although not all functions take 3 children and terminal functions don't have children, we can roughly calculate the upper bound of the search space by 9^(3^9), which is 2.3e+18782.

Program terminates either when an individual has been found that has 100% hit rate, or when the maximum number of generation has been reached. Below is the tableau for the graph coloring problem.

Table 3 Tableau for the graph coloring problem

Objective:	Find a graph coloring algorithm that is competitive with the currently best known coloring algorithms
Terminal Set:	ColorRand, ColorSafe, Decolor, Equalize, MakeUnique
Function Set:	IfColored, IfUnique, Progn2, Progn3
Fitness cases:	10 graphs selected from the DIMACS Challenge benchmarks that represent a variety of graph types.
Raw fitness:	The number of edges in the graph that connect two nodes of different color divided by the total number of edges in the graph.
Standardized fitness:	Sum over the 10 different graphs of the ratios between the number of incorrect edges and total number of edges.
Hits:	A hit means a graph has been colored using the no more than the best-known number of colors for the given graph.
Wrapper:	None
Parameters:	M = 500 G=50

5. Results

The best of run individual after 60 generations is able to color 6 out of 10 graphs with optimal coloring. Figure 2 presents the performance curves showing, by generation, the fitness value and hit count of the average and best individuals. The ideal individual should have fitness value 0 and hit count 10.

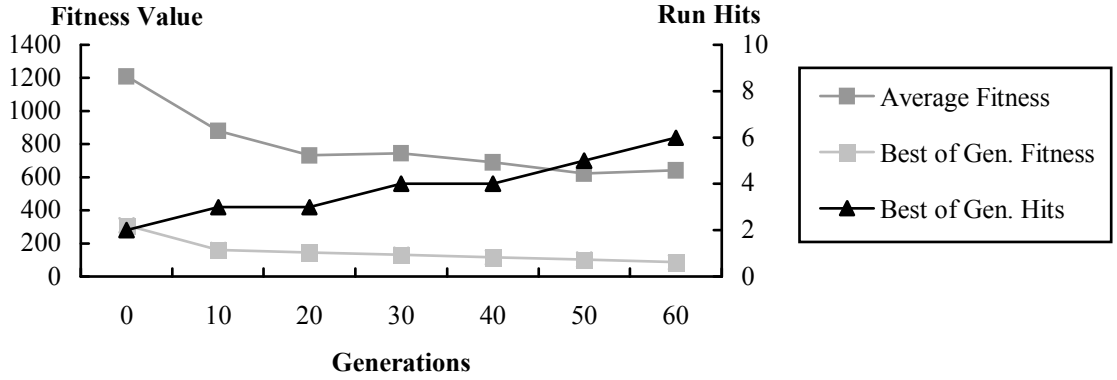

Figure 2 Performance curves for graph coloring algorithms

Table 5 Best of run individual statistics

Name	Uncolored Nodes / Total Nodes	Allowed Colors	Incorrect Edges / Total Edges	Hit
David.col	0 / 87	11	9 / 406	No
Huck.col	0 / 74	11	0/301	Yes
Jean.col	0 / 80	10	0 / 254	Yes
Myciel3.col	0 / 11	4	0 / 20	Yes
Myciel4.col	0 / 23	5	5 / 71	No
Myciel5.col	0 / 47	6	0 / 236	Yes
Queen5_5.col	0 / 25	5	6 / 160	No
Queen6_6.col	0 / 36	7	0 / 290	Yes
Queen7_7.col	0 / 49	7	21 / 476	No
Queen6_6.col	0 / 64	9	0 / 728	Yes

6. Discussion of Results

The result shows that the best of run individual after 60 generations is able to color 6 graphs out of 10, while having a lower number of incorrect edges in the rest. Also, a trend of improvement can be observed in the best-of-run individuals as generation increases. Although there is distance between the ideal individual and the best individual shown here, the results are still significant because near-optimal solution for the graph coloring problem is also a well-researched area due to its practical values. Further more, because the best-known numbers of the test cases are the collective best results from running all known algorithms, even the most competitive algorithm may not be able to find the best coloring for all graphs single handedly.

7. Conclusion

In this paper, we have presented a method of finding graph coloring algorithm via genetic programming. Because of the theoretical and practical values of the graph coloring problem, highly involved researches have gone into the studies of finding algorithms computing exact or approximate solutions. This paper demonstrates that by genetic programming, one can obtain fairly competitive algorithms without manually applying complicated optimizations. The paper also demonstrates that there is a potential that further improvements made in the five preparatory steps may lead to highly competitive individuals.

8. Future Work

Several improvements in the five preparatory steps of the graph coloring problem may worth further investigation. In particular, the terminal and function set can be evaluated and perhaps extended to allow more types of operations. In addition, with better available hardware, the population size and max generation can be increased to allow more diverse individuals evolving through longer periods. Finally, more benchmark graphs can be included in the test cases. This may help to evolve more individuals that have a better overall performance but maybe lacking in certain specialized graphs.

9. References

Garey, Michael R. 1999. *Computers and Intractability: A guide to the Theory of NP-Completeness*. W.H. Freeman & Company. Pages 84-90.

Johnson, David S. 1996. DIMACS Series Volume 26. Trick, Michael A.(editor). *Cliques, Coloring, and Satisfiability*. American Mathematical Society. Pages 245-255.

Culberson, Joseph and Gent, Ian P. 2001. Theoretical Computer Science. *Frozen Development in Graph Coloring*. Pages 228-230.

Cormen, Thomas H., Leiserson, Charles E., Rivest, Rondald. L. 1994. *Introduction to Algorithms*. Pages 962-963.

Hertz, A. and Werra, D. 1987. *Using Tabu Search Techniques for Graph Coloring' in Computing*. Pages 345-351.