Evolutionary Construction of Graph-Coloring Programs using Genetic Programming

Sarit Chantasuban and Ekawit Nantajeewarawat*
Information Technology Program
Sirindhorn International Institute of Technology, Thammasat University
P.O. Box 22, Thammasat-Rangsit Post Office,
Pathumthani 12121, Thailand

**Corresponding Author*

Abstract

Genetic programming (GP) is a novel paradigm that simulates the way of solving problems by nature according to Darwin's theory of fitness-driven natural selection. Instead of using bit strings as in genetic algorithm (GA), GP uses tree structures as its computing structures. As computer programs can be represented as trees, GP has been employed as a method of generating computer programs. In the work reported in this paper, GP is applied to the graph-coloring problem, an NP-complete problem which is an abstraction of many real-world practical problems, with expectation of constructing computer programs that are capable of computing approximations of the optimal solutions to many instances of the problem. The resulting computer programs are analyzed and their performance is compared with two existing commonly used approximation algorithms for graph coloring, i.e., the sequential coloring algorithms with random coloring order and with maximal-to-minimal-degree coloring order.

Keywords: Genetic programming, program synthesis, graph-coloring problem

1. Introduction

Various computing paradigms in artificial intelligence, including inductive learning, artificial neural networks, conventional genetic algorithms, etc., have been proposed as responses to one of the central questions in computer science: "How can computers learn to solve problems without being explicitly programmed? [12]" Computation in these paradigms is based on some specialized structures, such as flat records of observable properties, vectors of weights, chromosome strings (bit strings), etc. However, such specialized structures are often unnatural and impose some constraints on solving problems [5]. Based on the basic idea that tree structures are more flexible and provide more expressive power, the genetic programming (GP) paradigm [2,5] has emerged as a promising way of finding problem solutions encoded as trees. As computer programs themselves can be represented as trees, GP has been applied in diverse fields as a method of evolving computer programs [6,7,8], it is also known as evolutionary programming.

While a conventional genetic algorithm (GA) is typically used when the solution to an instance of a problem can be easily represented as a value or a set of values, GP is more often used when the solution is a particular behavior or function. In this paper, GP is applied to the *graph-coloring problem*, an NP-complete problem, the optimal solutions to instances of which can, in general not be determined by means of any known deterministic algorithm in polynomial time. In contrast with most of the applications of GP, which attempt either to directly approximate the optimal solution to some specific instance of a problem or to generate a computer program for solving some specific problem instance, this paper aims at studying the possibility of applying GP to the construction of computer programs that can be used for solving the graph-coloring problem in general. Two groups of experiments have been conducted. The per-

formance of the constructed computer programs is investigated by comparing them with existing widely used approximation algorithms for graph coloring, i.e., the sequential coloring algorithm with random coloring order and the sequential coloring algorithm with maximal-to-minimal-degree coloring order.

The rest of this paper is organized as follows. Section 2 recalls the graph-coloring problem along with some of its practical examples. Section 3 presents the experimental results and analyzes the performance and the internal structures of the generated programs. Section 4 discusses some related works and draws conclusions.

2. The Graph-Coloring Problem

The graph-coloring problem is an NP-complete problem – one of the hardest problems in the class NP (non-deterministic polynomial problems). The problem can be formalized as follows. An assignment of colors to a graph $G = (V, E)$ is a mapping $C: V \rightarrow S$, where S is a finite set of colors, which are normally represented as integers, such that if $(v, w) \in E$ then $C(v) \neq C(w)$; in other words, the same color is not assigned to adjacent vertices. Given a graph, the problem is to find the *minimum number of colors needed* for coloring the graph according to the above requirement. This minimum number is also referred to as the **chromatic number** of the given graph.

The graph-coloring problem is an abstraction of many real-world problems [1], such as time-table scheduling, radio-frequency assignment, registers allocation in compiler construction, and printed circuit board testing, etc. For example, in scheduling lecture classes at a school, each lecture class can be represented as a vertex, each lecture-time period as a color, and a conflict between two lecture classes that cannot be scheduled at the same time as an edge, and then, the chromatic number will correspond to the minimum number of lecture-time periods required. Likewise, in assignment of program variables to hardware registers during program compilation, where one variable is considered to be in conflict with another variable if the former is used, both before and after the latter within a short period of execution time, each variable can be represented as a vertex, each hardware register as a color, and a conflict between two variables as an edge, and, consequently, the chromatic number will indicate the minimum number of registers needed. In testing printed circuit boards for unintended short circuits, a net on a board can be viewed as a vertex, a possibility of a short circuit between two nets as an edge, and, then, an assignment of colors will partition the nets into supernets, which can be simultaneously tested for existence of short circuits, thereby speeding up the testing process.

As the problem is NP-complete, there is no known deterministic algorithm that can always find the chromatic number of a graph in polynomial-bounded time. In practice, heuristic algorithms have been employed for approximating the chromatic numbers of graphs.

Sequential Coloring Algorithms

Several heuristic algorithms for graph coloring have been proposed [9]. Among them, the class of sequential coloring algorithms (SC algorithms) is best known and most widely used. An algorithm in this class can be described as follows:

Algorithm: SC Algorithm
Input: a graph G.
Output: the chromatic number K of G.
1. [Initialization] $K := 0$.
2. [Cycle] While there is an uncolored vertex in G, repeat the following three steps:
2(a) [Choose a vertex] Choose an uncolored vertex v in G.
2(b) [Determine the color] Find the minimum integer b such that no vertex adjacent to v has already been colored by b.
2(c) [Color the chosen vertex] Color v by b, and if $K < b$, then $K := b$.

The order of vertices to be chosen in Step 2 (a) above characterizes the SC algorithm. If a vertex is randomly selected from the set of uncolored vertices, the algorithm is called the *SC algorithm with random coloring order*. If a vertex with maximal degree[1] is selected from the set of uncolored vertices in that step, the algorithm is called the *SC algorithm with maximal-to-minimal-degree coloring order*. For short, in the sequel, these two algorithms will be referred to as the **Random SC** algorithm and the **Max-min SC** algorithm, respectively.

[1] The degree of a vertex is the number of edges associated with the vertex.

3. Experiments

Two groups of experiments, i.e., Groups A and B, have been conducted using *lilgp* version 1.1 genetic programming system [13] running on Sun Ultra-10 workstations. Each group comprises four experiments. The experiments in the same group use the same problem formulation and control parameters but different sets of fitness cases.

3.1 Group-A Experiments

There are four experiments in Group A, i.e., Experiments I, II, III and IV. The problem formulation, specifying the set of terminals, the set of functions as well as the fitness measure, and the control parameters used in these experiments are described below.

Problem Formulation

Terminal Set: Seventeen primitive operations, listed in Table 1, are used as terminals. Most of them are operations for traversing graphs; for example, move from the current vertex to an unvisited vertex having maximal degree ('`mv-gmx-nvsted`'), and move to an uncolored vertex adjacent to the current vertex ('`mv-ncolred`'), etc.

Function Set: Functions are operators that appear as interior nodes in program trees. Six functions, listed in Table 2, are used. They are determined by basic control-flow statements, i.e., sequential, conditional, and iterative statements.

Fitness Measure: In each generation of a GP run, the fitness of an evolved program is evaluated by testing the program with a set of graphs, called *fitness cases*. Each set of fitness cases consists of 100 randomly generated graphs having the same edge-density value.[2] Each graph is connected, i.e., each vertex in the graph is adjacent to some other vertex, and contains at most 100 vertices. The edge-density values of the graphs in the sets of fitness cases used in Experiments I, II, III, and IV are 0.1, 0.2, 0.5 and 0.8, respectively; and the four sets of fitness cases are referred to as FC-0.1, FC-0.2, FC-0.5 and FC-0.8, respectively.

Each program in an experiment is tested with all graphs in the set of fitness cases for that experiment, one graph at a time. Testing a program P with a graph G terminates when P has colored all vertices in G or the amount of time P has spent exceeds the time limit specified by the maximum-evaluation-time control parameter. The evaluation time is estimated using the number of applications of primitive operations. That is, the execution of each terminal is assumed to take one logical time unit, while the execution of a function, which is typically a control-flow statement, is assumed to take zero time. This evaluation time is merely an approximation, and is not directly used in determining the performance of a program.

As the primary goal of a graph-coloring program is to find the minimum number of colors required, the performance of a program is measured by the number of colors it uses, i.e., a program using fewer colors has better performance. Accordingly, some fitness measure terms are defined as follows. The **raw fitness**, f_r, is the total number of colors[3] used in coloring all graphs in a set of fitness cases within the maximum fitness evaluation time. If some graph is not successfully colored, some penalty, i.e., the number of uncolored vertices times the constant 10, will be added to the raw fitness. In this problem, since the raw fitness should be minimized, the raw fitness is also used as the **standardized fitness**, f_s. As usual, the **adjusted fitness**, f_a, is derived from the standardized fitness by $f_a = (1 + f_s)^{-1}$. The number of **hits** is the number of graphs that are colored successfully (no uncolored vertex left) in a set of fitness cases.

Control Parameters

The major control parameters specified for a GP run in each experiment are as follows: the maximum number of generations is 1001. The population size is 500. The depth of a program tree in the initial population is between 2 and 6. The maximum depth of a program tree is 17. The probability of crossover is 0.9; the probability of internal crossover point is 0.9; the probability of external crossover point is 0.1;

[2] The edge-density value of a graph G having n vertices is the ratio of the number of edges in G to the maximum number of edges that a graph with n vertices can contain. That is, if the number of edges in G is N_E, then the edge-density value of G is $2N_E/n(n\text{-}1)$.

[3] Since the main task of an evolved program is to approximate the chromatic number of a graph, the colors used in coloring different graphs are considered to be all different.

and the probability of reproduction is 0.1. In order to generate program trees having a wide variety of sizes and shapes, the ramped half-and-half method [5] is used for generating the initial population. The maximum fitness evaluation time is specified as 30,000 logical time units.

Benchmark Graphs

Twelve graphs, the chromatic numbers of which are known beforehand, are used as benchmarks for comparing the performance of evolved programs with the Random SC algorithm and the Maxmin SC algorithm. These twelve benchmark graphs, which are taken from [10], are characterized in Table 3.

Group-A Experimental Results

Table 4 shows the total number of colors used by the Random SC algorithm and the Maxmin SC algorithm when they are tested with the four sets of fitness cases and the twelve benchmark graphs, whereas Table 5 shows the performance of the best-so-far evolved programs[4] of the experiments in Group A. The tables indicate that for their respective sets of fitness cases, the best-so-far evolved programs perform better than both the Random SC algorithm and the Maxmin SC algorithm. However, when they are tested with the benchmark graphs, the best-so-far evolved programs of Experiments I and III fail to color the benchmark graphs entirely due to the specified time constraint, while those of Experiments II and IV yield satisfactory results in comparison with the two SC algorithms, i.e., they use 40 and 43 colors, respectively, fewer than the Random SC algorithm, and use only 7 and 4 colors (0.0276% and 0.0159%), respectively, more than the Maxmin SC algorithm.

Figure 1 shows the adjusted fitness curves of the GP runs in the experiments in Group A. As signified by the figure, the development of the evolved programs in these experiments is unsatisfactory in comparison with a normal GP run. In particular, the adjusted fitness curves of the GP runs in Experiments I, II, and III exhibit wide swings and seem not to converge to any particular value.

3.2 Group-B Experiments

In order to improve the development of evolved programs, and to enhance the performance of the generated programs, especially the programs obtained from Experiments I and III, the set of functions and the set of terminals are revised, and another group of experiments, i.e., Group B, has been conducted. This group consists of Experiments V, VI, VII and VIII. The sets of fitness cases used in Group A are also employed in this group, i.e., the sets FC-0.1, FC-0.2, FC-0.5 and FC-0.8 of fitness cases are used in Experiments V, VI, VII and VIII, respectively. The same set of benchmark graphs (see Table 3) are used.

Revised Problem Formulation

Using the problem formulation of Group A, all the evolved programs that succeed in assigning colors to graphs contain at least one occurrence of the terminal '`color`', and most of them have at least one occurrence of the function '`for-all-vertces`'. The terminal '`color`' must be applied to each vertex in a graph at least once, otherwise assignment of colors will not be complete. The function '`for-all-vertces`' is also necessary for traversing all vertices in a graph.[5] From this observation and the unsatisfactory development of programs in the experiments in Group A, the authors hypothesize that evolutionary construction of computer programs should not depend on any particular necessary terminal and/or function, otherwise the structures of evolved programs will be constrained to have some specific content and the variety of evolved programs will be reduced. Based on this hypothesis, the terminal '`color`' and the function '`for-all-vertces`' are identified to be inappropriate.

Accordingly, the set of terminals and the set of functions are revised. The terminal '`color`' is removed, and its operation (i.e., assigning the minimum possible color to the current vertex) is instead incorporated into the traversing-related terminals. For example, the traversing-related terminal '`mv-gmx-ncolred`' is now modified in such a way that after moving to an uncolored vertex with globally maximal degree, the mini-

[4] In this paper, the best-so-far evolved program of an experiment is the program of which the raw fitness is not higher than any other evolved program in any generation in that experiment.

[5] Although a cascade of '`for-all-neighbs`' is a possible alternative, it would result in a large program structure.

mum possible color will be assigned to the current vertex. All other traversing-related terminals are also modified in this way. After such modification, some terminal now has exactly the same operation as another, e.g., since a vertex will now always be colored whenever it is visited, the terminal 'mv-gmx-ncolred' and 'mv-gmx-nvsted' perform the same task. Terminals with duplicate operations are also removed. The resulting terminal set consists of 10 terminals, i.e., those marked with 'x' in the third column of Table 1.

Based on the hypothesis, the function 'for-all-vertces' is also removed from the function set. Instead, it is assumed that every evolved program will be executed iteratively by some external control mechanism until there is no uncolored vertex left. With such implicit external control mechanism, the function 'for-all-neighbs' can also be removed. Furthermore, the functions 'if-colred' and 'if-ncolred' are now unnecessary due to the revision of the terminal set[6] and are also removed. The revised set of functions, used in Group B, therefore contains only two functions, i.e., 'prog2' and 'prog3'.

It is worth remarking that the removal of the function 'for-all-vertces' has another merit. This function consumes a considerable amount of time in fitness evaluation since its argument is applied to all vertices in the graph. In particular, when an evolved program contains several occurrences of this function and a given graph has many vertices, the evolved program is likely to fail to color the given graph within the specified time limit. With the revised set of functions, an evolved program tends to require much less time to color a graph. The maximum fitness evaluation time is thus reduced from 30,000 logical time units for Group A to 3,000 logical time units for Group B. Other control parameters are unchanged.

Group-B Experimental Results

Figure 2 shows the adjusted fitness curves of the GP runs in the Group-B experiments, and Table 6 shows the total number of colors used by the best-so-far programs of these experiments in coloring their respective sets of fitness cases and the benchmark graphs. They indicate that the results obtained from the GP runs in Group B are better than those in Group A in terms of both the development of evolved program trees and their performance.

As demonstrated by Tables 4 and 6, the performance of the best-so-far evolved programs of the experiments in Group B is significantly better than that of the Random SC algorithm, i.e., the best-so-far evolved programs of Experiments V, VI VII and VIII use 67, 91, 124 and 168 colors, respectively, fewer in coloring the graphs in their respective sets of fitness cases, and use 31, 43, 43, and 39 colors, respectively, fewer in coloring the benchmark graphs. In comparison with the Maxmin SC algorithm, these best-so-far programs have slightly better performance in coloring their respective sets of fitness cases, i.e., they use 12, 15, 18 and 5 colors, respectively, fewer. However, when tested with the benchmark graphs, the performance of the Maxmin SC algorithm is slightly better, i.e., the Maxmin SC algorithm uses 16, 4, 4 and 8 colors fewer (0.0608%, 0.0159%, 0.0159% and 0.0314% better) than the best-so-far programs of Experiments V, VI, VII and VIII, respectively, in coloring the twelve benchmark graphs entirely. Notwithstanding, it should be remarked that, as will be seen in the next subsection, evolved programs with exactly the same performance as the Maxmin SC algorithm are also obtained from some experiments in Group B.

3.3 Analysis of Evolved Programs

To illustrate some program trees obtained from the GP runs in Group B, the best-so-far programs of Experiment V (obtained from Generation #415) and Experiment VI (obtained from Generation #741) are shown in Figures 3 and 4, respectively.[7] When tested with the benchmark graphs, the performance of the program tree in Figure 4 is better than that of the program tree in Figure 3 (it uses 12 colors fewer). Observe that the majority (13 out of 19 or 68.4%) of the terminals in the program tree in Figure 4 is 'mv-gmx-ncolred'.

In Experiments VI and VIII, GP also generates several programs that have almost the

[6] Using the revised set of terminals, the current vertex will always be colored.

[7] In the experiments, program trees are represented as S-expressions. For the sake of readability, however, the programs trees illustrated in this paper are presented in a graphical form.

same or exactly the same performance as the Maxmin SC algorithm when tested with the twelve benchmark graphs. Such programs include the fourth best programs of Experiment VI, shown in Figures 5 and 6, and the second best programs of Experiment VIII, shown in Figures 7 and 8.

A closer examination of the results reveals that evolved programs which produce good coloring results for the benchmark graphs in comparison with the Maxmin SC algorithm (such as those in Figures 4, 5, 6, 7 and 8) mainly use the terminal 'mv-gmx-ncolred' as their coloring strategy. As the Maxmin SC algorithm uses the terminal 'mv-gmx-ncolred' solely as its traversing operation, these programs can be considered as variants of the Maxmin SC algorithm. It should be noted that in company with some other terminals, such as 'mv', 'mv-max', and 'mv-vsted', some of these variants perform even better than the Maxmin SC algorithm for their respective sets of fitness cases. For example, the programs in Figures 5 and 6 both use 619 colors, while the Maxmin SC algorithm uses 625 colors, in coloring the graphs in the set FC-0.2 of fitness-cases. Moreover, some program trees that work exactly in the same way and have exactly the same performance as the Maxmin SC algorithm are also generated, e.g., the program tree in Figure 8, which is constructed in Generation #941 of Experiment VIII.

4. Related Works and Conclusions

Both GP and the conventional genetic algorithm (GA) simulate the way of solving problems by nature according to Darwin's theory of fitness-driven natural selection, i.e., fitter individuals survive and reproduce at a higher rate than other individuals. GA, however, employs passive chromosome strings (bit strings) as its underlying computing structures, and a chromosome string obtained from GA can merely represent a solution to a single specific instance of a problem. In the graph-coloring problem, for example, an entire process of GA run is required in order to approximate the chromatic number of an individual graph. By contrast, GP uses more flexible and more powerful computing structures, which may represent either a direct solution to an instance of a problem or a program tree for solving some problem instance. As a program tree is, in a sense, an active structure, it may also be possible to gradually construct by means of GP a general computer program that can be used for solving some class of problem instances.

In [3,4,11], GP has been employed as a method of approximating the optimal solutions to NP problems. Nonetheless, in these works, an approximate solution is specific to only a particular problem instance. For example, in [3], GP is applied to the facility layout problem (FLP), which is an NP-complete combinatorial optimization problem, in order to generate a slicing tree structure (STS) representing an approximation of the optimal arrangement of a given collection of facilities. Such a slicing tree is, however, a passive binary tree, in which each terminal node simply represents a facility and each interior node expresses the relation between its children substructures, and does not represent a computer program. In [4,11], GP is employed to solve the minimum clique problem, another NP-complete problem, by generating a program tree for finding the largest complete subgraph of a given graph, but the resulting program tree is specifically tailored for the given graph only.

The primary objective of the effort reported in this paper is to study the possibility of applying GP to the construction of general computer programs for solving the graph-coloring problem. Since the resulting programs are expected not to be specific to any particular graph (instance of the problem), a set of fitness cases is used in each experiment instead of a single fitness case as in usual GP applications. After the revision of the problem formulation of the Group-A experiments, the best-so-far evolved programs of the experiments in Group B can successfully color all the benchmark graphs. In terms of performance, the best-so-far evolved programs yield far better approximations of the chromatic numbers of both the graphs in their respective sets of fitness cases and the benchmark graphs than the Random SC algorithm, and produce approximations which are comparable to those computed by the Maxmin SC algorithm. In particular, as seen in Subsection 3.3, if one takes not only the best-so-far program of a particular experiment, but a set consisting of a few top programs (for example, the top three programs) of each experiment in Group B as the result of program construction, then the best coloring results obtained from the programs in this set are at least as good as the results ob-

tained from the Maxmin SC algorithm, both for the sets of fitness cases and the benchmark graphs.

Close analysis of the internal structures of the constructed programs shows that the program trees that either are variants of, or work in the same way as the Maxmin SC algorithm bear good coloring results, especially for the benchmark graphs. To the authors' knowledge, the Maxmin SC algorithm is the best human-known graph-coloring algorithm that can be constructed out of the terminals and functions used in the experiments. This work therefore provides evidence supporting the application of GP to computer program synthesis. Given a new problem for which no known algorithm already exists and a set of basic operations and functions that are supposed to be used, GP can be fairly expected to gradually construct good computer programs for solving at least some class of instances of the problem.

More evidence supporting such application of GP may be gained by conducting further experiments with some other well-known problems, such as the bin-packing problem, the knapsack problem, the Hamiltonian cycle problem, and the traveling salesperson problem [1,9], and comparing the internal structures and the performance of resulting computer programs with human-known algorithms that can be derived from the terminals and functions used in their respective experiments.

5. References

[1] Baase, S., *Computer Algorithms: Introduction to Design and Analysis*, pp. 320-354, Addison, 1998.

[2] Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D., *Genetic Programming-An Introduction: On the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufmann Publisher, 1998.

[3] Garces-Perez, J., Schoenefeld A. D., and Wainwright L. R., Solving Facility Layout Problems Using Genetic Programming, *Proceedings of the First Annual Genetic Programming Conference*, Cambridge, The MIT Press, 1996.

[4] Haynes, T., and Schoenefeld D., Clique Detection via Genetic Programming, *Proceedings of the First Annual Genetic Programming Conference*, Cambridge, The MIT Press, 1996.

[5] Koza, J. R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: The MIT Press, 1992.

[6] Koza, J. R., et al., editors, *Proceedings of the First Annual Genetic Programming Conference*, Cambridge, The MIT Press, 1996.

[7] Koza, J. R., et al., editors, *Proceedings of the Second Annual Genetic Programming Conference*, San Francisco, Morgan Kaufmann, 1997.

[8] Koza, J. R., et al., editors, *Proceedings of the Third Annual Genetic Programming Conference,* Wisconsin, Morgan Kaufmann, 1998.

[9] Kucera, L., *Combinatorial Algorithms*, pp. 215-219, New York, Adam Hilger, 1990.

[10] Leighton, F. T., Journal of Research of the National Bureau of Standards, 84: 489-505, 1979.

[11] Soule, T., Foster A. J., and Dickinson J., Using Genetic Programming to Approximate Maximum Clique, *Proceedings of the First Annual Genetic Programming Conference*, Cambridge, The MIT Press, 1996.

[12] Samuel, A. L., Some Studies in Machine Learning Using the Game of Checkers, *IBM Journal of Research and Development* 3(3): 210-229, 1959.

[13] Zongker, D., and Punch, B., *lil-gp 1.0 User's Manual*, 1995. www.garage.cps.msu.edu/software/software-index.html.

Terminal	Group A	Group B	Description
color	x	-	Assign the minimum possible color to the current vertex.
mv	x	x	Move to the first adjacent vertex.
mv-gmx-ncolred	x	x	Move to a globally maximal degree, uncolored vertex.
mv-gmx-nvsted	x	-	Move to a globally maximal degree, unvisited vertex.
mv-gmx-vsted	x	x	Move to a globally maximal degree, visited vertex.
mv-gmx-vsted-ncolred	x	-	Move to a globally maximal degree, visited but uncolored vertex.
mv-gmn-ncolred	x	x	Move to a globally minimal degree, uncolored vertex.
mv-gmn-nvsted	x	-	Move to a globally minimal degree, unvisited vertex.
mv-gmn-vsted	x	x	Move to a globally minimal degree, visited vertex.
mv-gmn-vsted-ncolred	x	-	Move to a globally minimal degree, visited but uncolored vertex.
mv-max	x	x	Move to an adjacent vertex with maximal degree.
mv-min	x	x	Move to an adjacent vertex with minimal degree.
mv-ncolred	x	x	Move to an uncolored, adjacent vertex.
mv-nvsted	x	-	Move to an unvisited, adjacent vertex.
mv-vsted	x	x	Move to a visited, adjacent vertex.
mv-vsted-ncolred	x	-	Move to a visited but uncolored, adjacent vertex.
nop	x	x	No-operation.

Table 1: Terminals
The terminals in the entries marked with 'x' in the second and the third columns are used in the Group-A experiments and the Group-B experiments, respectively.

Function	No. of Arguments	Group A	Group B	Description
for-all-neighbs	1	x	-	Apply the argument to all adjacent vertices.
for-all-vertces	1	x	-	Apply the argument to all vertices.
if-colred	2	x	-	Apply the first argument if the current vertex is colored; otherwise, apply the second argument.
if-ncolred	2	x	-	Apply the first argument if the current vertex is uncolored; otherwise, apply the second argument.
progn2	2	x	x	Apply the two arguments consecutively.
progn3	3	x	x	Apply the three arguments consecutively.

Table 2: Functions
The functions in the entries marked with 'x' in the second and the third columns are used in the Group-A experiments and the Group-B experiments, respectively.

Graph #	Edge Density	No. of Nodes	No. of Edges	Chromatic Number
1	0.080851	450	8168	15
2	0.080861	450	8169	15
3	0.165108	450	16680	15
4	0.165801	450	16750	15
5	0.081762	450	8260	25
6	0.081792	450	8263	25
7	0.171670	450	17343	25
8	0.172482	450	17425	25
9	0.056560	450	5714	5
10	0.056758	450	5734	5
11	0.097035	450	9803	5
12	0.096580	450	9757	5

Table 3: Characteristics of the Benchmark graphs

Graphs	Random SC	Maxmin SC
FC-0.1	497	442
FC-0.2	701	625
FC-0.5	1285	1179
FC-0.8	2139	1976
Benchmarks	294	247

Table 4: Performance of the two SC algorithms

Graphs	Exp. I	Exp. II	Exp. III	Exp. IV
FC-0.1	433	-	-	-
FC-0.2	-	615	-	-
FC-0.5	-	-	1172	-
FC-0.8	-	-	-	1973
Benchmarks	FAILED	254	FAILED	251

Table 5:
Performance of the best-so-far programs in Group A

Graphs	Exp. V	Exp. VI	Exp. VII	Exp. VIII
FC-0.1	430	-	-	-
FC-0.2	-	610	-	-
FC-0.5	-	-	1161	-
FC-0.8	-	-	-	1971
Benchmarks	263	251	251	255

Table 6:
Performance of the best-so-far programs in Group B

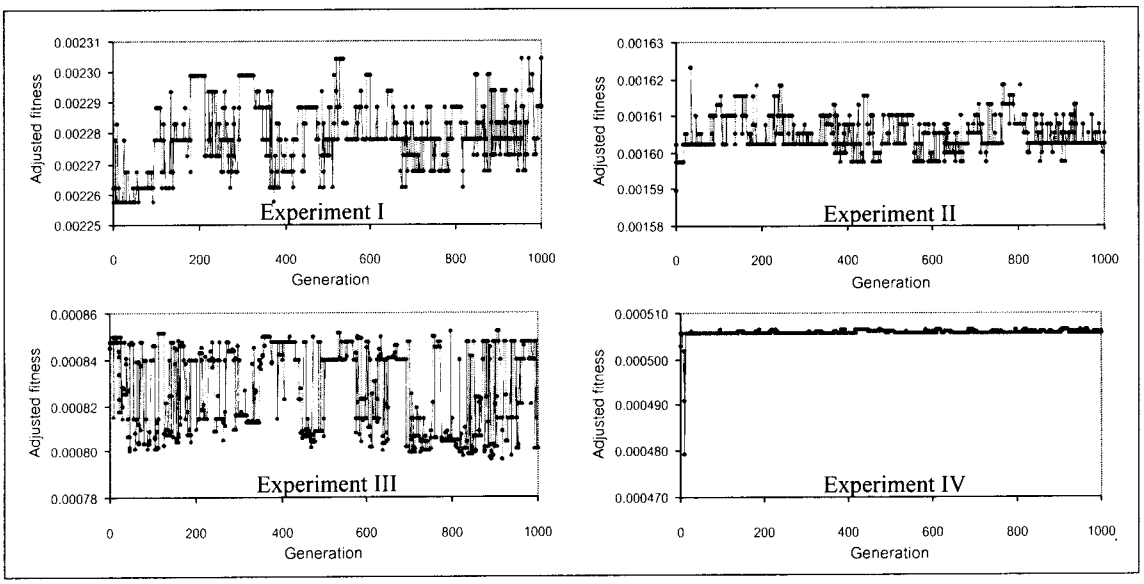

Figure 1: Adjusted fitness of the best-of-generation programs of the Group-A experiments

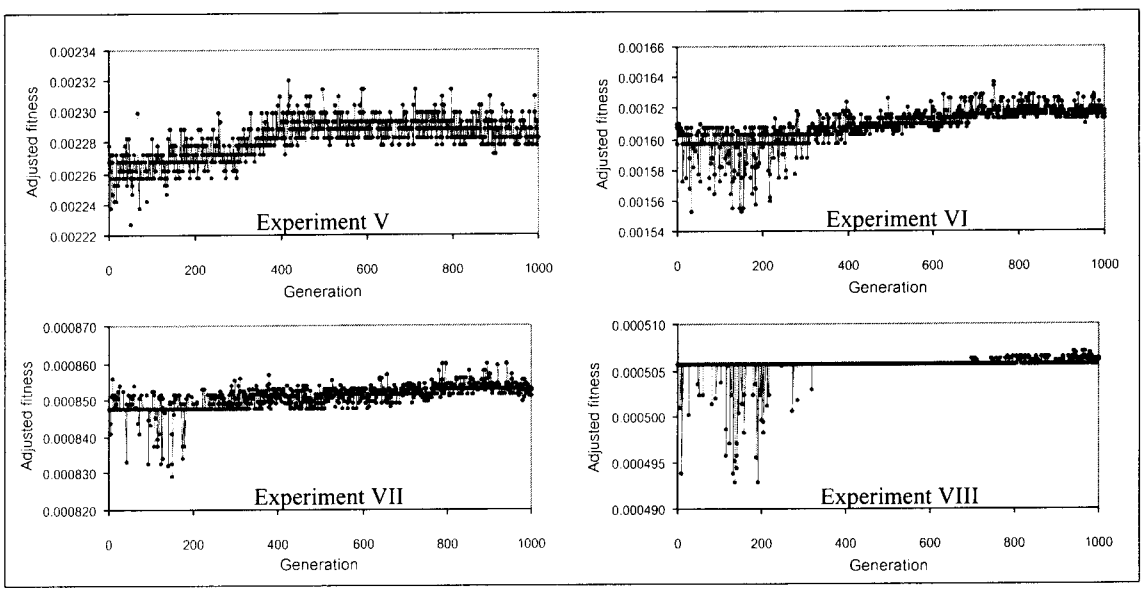

Figure 2: Adjusted fitness of the best-of-generation programs of the Group-B experiments

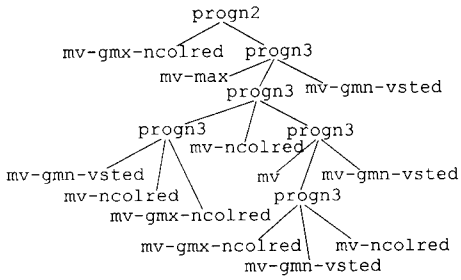

Figure 3:
The best-so-far program of Experiment V
(Raw fitness = 430; 263 colors for the benchmarks)

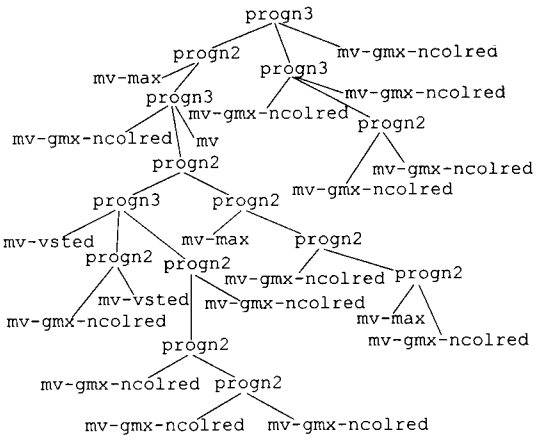

Figure 4:
The best-so-far program of Experiment VI
(Raw fitness = 610; 251 colors for the benchmarks)

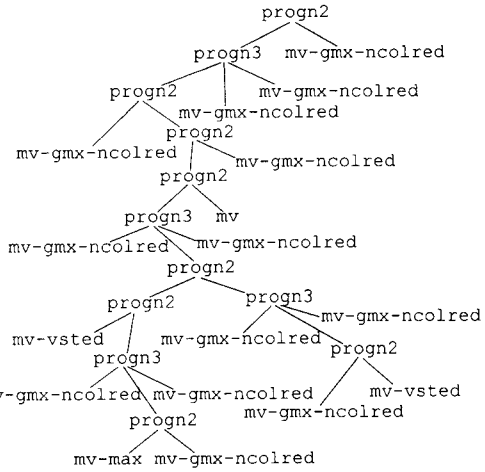

Figure 5:
One of the 4th best programs of Experiment VI
(Raw fitness = 619; 248 colors for the benchmarks)

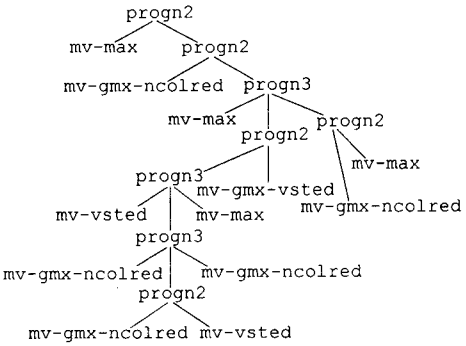

Figure 6:
One of the 4th best programs of Experiment VI
(Raw fitness = 619; 247 colors for the benchmarks)

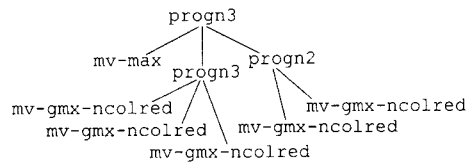

Figure 7:
One of the 2nd best programs of Experiment VIII
(Raw fitness = 1976; 248 colors for the benchmarks)

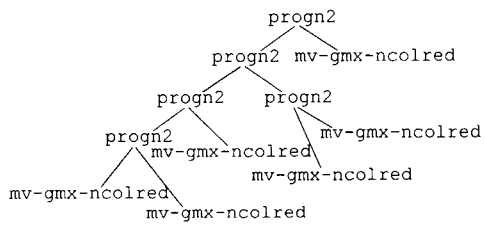

Figure 8:
One of the 2nd best programs of Experiment VIII
(Raw fitness = 1976; 247 colors for the benchmarks)