A GROUPING GENETIC ALGORITHM FOR COLORING THE EDGES OF GRAPHS

SAMI KHURI
Dept of Math & Computer Science
San José State University
San José, CA
khuri@cs.sjsu.edu

TIM WALTERS
Virage, Inc.
San Mateo, CA
t.walters@computer.org

YANTI SUGONO
2017 Paseo Del Sol
San José, CA
ysugono@hotmail.com

Keywords: *combinatorial optimization, edge coloring problem, genetic algorithms, grouping genetic algorithm*

ABSTRACT

This paper investigates applications of heuristic techniques for solving the edge coloring problem, which seeks the minimum number of colors to color the edges of a graph such that no two adjacent edges (edges that share a common vertex) get the same color. Except for special cases, such as bipartite graphs, the edge coloring problem is NP-complete. Thus, the search for exact algorithms is replaced by the investigation of approximation and heuristic algorithms (unless P=NP). In this work, we consider both simple graphs and multigraphs, and compare three heuristics for the edge coloring problem. The first is a greedy algorithm, and the two others are genetic algorithms: a genetic algorithm that makes use of LibGA and a grouping genetic algorithm. Our results show that the grouping genetic algorithm outperforms the greedy heuristic for many problem instances. This paper supports the claim that the edge coloring problem is more amenable to grouping genetic algorithms.

1 INTRODUCTION

The edge coloring problem seeks to label a graph's edges with a minimum number of colors in such a way that no two edges that share a vertex also share a color. The minimum number of colors required to color the edges of a graph G is called the chromatic index of G and is generally denoted by $\chi'(G)$. It should not be confused with the minimum number of colors to color the vertices of G, the chromatic number, which is usually denoted by $\chi(G)$. Holyer proved that the edge coloring problem is NP-hard [9]. Thus, the search for finding exact algorithms for the edge coloring problem should be replaced by designing efficient approximate algorithms and heuristic algorithms. In this work, we investigate and compare two genetic algorithms (GAs) with a greedy heuristic.

We apply these heuristics not only to simple graphs but also to multigraphs: graphs that allow more than one edge between two vertices. We note that on multigraphs, the edge coloring problem can be approximated within $1.1 + (0.8/opt)$ [13].

The edge coloring problem has many practical applications including various scheduling problems, partitioning problems [7], timetabling problems [2], and wavelength-routing in optical networks [14].

We design and implement two genetic algorithms; the first makes use of LibGA [3] and the second is a Grouping Genetic Algorithm [6]. We also implement a greedy heuristic. Experimental results obtained by the GAs and the greedy heuristic on 28 simple graphs and 15 multigraphs are presented. Some graphs are taken from Knuth's Stanford GraphBase [12], and are part of the DIMACS's (Center for Discrete Mathematics and Theoretical Computer Science) benchmark graphs [4]. The other graphs were generated by us.

The rest of this work is organized in the following fashion. Section 2 defines the edge coloring problem. Section 3 introduces the greedy heuristic. The two GA implementations for the edge coloring problem are given in Section 4. The implementation for the GGA, and all stochastic operators used in both GAs, including the fitness function, are discussed in this section. Section 5 describes the test cases used in this work, as well as the results and analysis of the performance of the different heuristic algorithms. Section 6 contains our concluding remarks.

2 THE EDGE COLORING PROBLEM

As mentioned earlier, determining the chromatic index of G, denoted by $\chi'(G)$, is NP-hard [9]. As a matter of fact, even the restricted case of deciding whether a simple graph has chromatic index of *three or less* is NP-complete. As one might expect, $\chi'(G)$ is related to the maximum vertex degree of G, denoted by Δ. If G is a bipartite graph, then $\chi'(G) = \Delta$ [2]. If G is a nonbipartite, simple graph, then either $\chi'(G) = \Delta$ or $\chi'(G) = \Delta + 1$ [16]. This fact makes the edge coloring problem unique. It belongs to a handful of problems that are known as "within one unit of optimum" [8]. Indeed, the fact that the optimum solution is one of two quantities, Δ or $\Delta + 1$, makes it easy to evaluate the efficiency of the heuristic being studied. An efficient heuristic has to consistently produce a number of colors in the neighborhood of (not much larger than) the graph's maximum vertex degree.

The following formal definition of the edge coloring problem uses Stinson's terminology for combinatorial optimization problems [15]. We introduce concepts and notations that are used in subsequent sections.

Problem instance:

A graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ is the set of vertices and $E = \{e_1, \ldots, e_m\}$ the set of edges.
We denote by $Adj(e_i)$ the set of edges adjacent to e_i, i.e., the set of edges in G that share a common endpoint with e_i.

Feasible solution:

A feasible solution is represented by a vector $\vec{y} = y_1\, y_2\, \ldots\, y_m$ where $y_i = j$ means that the i^{th} edge e_i is given color j, $1 \leq j \leq m$. The components of $\vec{y}$ are essentially the colors assigned to the m edges of G.
We assume that the smallest value of any y_i is 1 and that if $q = max\{y_i \mid 1 \leq i \leq m\}$, then for any $t, 1 \leq t \leq q$, there exists at least one $k, 1 \leq k \leq m$ such that $y_k = t$. In other words, the colors are denoted by $1, 2, \ldots, q$.
We also have:

$$e_j \in Adj(e_i) \Rightarrow y_i \neq y_j$$

In other words, two adjacent edges have different colors.

Objective function:

For a feasible vector $\vec{y}$, let $P(\vec{y})$ be the number of colors used in the coloring of the edges represented by $\vec{y}$. More precisely,

$$P(\vec{y}) = max\{y_i \mid y_i \text{ is a component of } \vec{y}\}.$$

Optimal solution:

A feasible vector $\vec{y}$ that yields the smallest $P(\vec{y})$.

3 A GREEDY HEURISTIC FOR THE EDGE COLORING PROBLEM

EdgeColoringAssignment is a greedy algorithm that randomly chooses an edge e_i to color. To make sure that e_i gets a different color from its adjacent edges, the algorithm makes use of $Adj(e_i)$. Recall that $Adj(e_i)$ represents the set of edges adjacent to e_i, for $i = 1, 2, \ldots, m$. To color e_i, *EdgeColoringAssignment* checks the colors of every edge in $Adj(e_i)$ and gives e_i the smallest numbered color not already assigned to any of e_i's adjacent edges. Let color$[e_i]$ be the color number assigned to edge e_i, and *assignedcolorset* be the set of colors of the edges in $Adj(e_i)$ that have already been colored.

Algorithm EdgeColoringAssignment:
Randomly choose an edge, say e_i, and assign it color number 1; $color[e_i] = 1$;
for $i := 2$ to m
{
  assignedcolorset:= $\emptyset$;
  randomly select an edge that has not been colored, say e_j;
  check each member in $Adj(e_j)$;
    if it is already colored then add that number to *assignedcolorset*;
  $k := 1$;
  while k is in *assignedcolorset*
    $k := k + 1$;
  $color[e_j] = k$; //k is the smallest unassigned color.
}

The running time of the above algorithm is $O(m\Delta)$.

4 THE GENETIC ALGORITHMS

We start by discussing our implementation of the GA that uses LibGA, and then turn our attention to the grouping genetic algorithm. Some features, such as the fitness function, are the same for both GAs.

4.1 Representation and GA Operators

A coloring of a graph's edges is represented by using a non-binary string, $y_1\, y_2\, \ldots\, y_m$, where m is the number of edges. The value of y_j represents the color assigned to the j^{th} edge. If a graph has 12 edges, for example, then the following string is a possible coloring, where edge e_1 has color number 5, edge e_2 has color number 3, and so on.

1	2	3	4	5	6	7	8	9	10	11	12
5	3	5	5	1	3	2	1	4	4	1	3

Figure 1: Chromosome representation for the edge coloring problem

In our study, we use generational genetic algorithms with roulette wheel selection, uniform crossover, and uniform mutation for both genetic algorithms. Each string in the population occupies a slot on the roulette wheel of size *inversely* proportional (since the edge coloring problem is a minimization problem) to the fitness value.

Unlike many other approaches that handle constrained optimization problems by a knowledge–based restriction of the search space to feasible solutions, the GA described here uses a penalty function to handle constraints. Infeasible strings join the population, but at a price. A penalty term reduces an infeasible string's fitness. The fitness function obeys the following principle [10, 11]:

The fitness functions use graded penalty functions. Two infeasible strings are not treated equally. The penalty is a function of the distance from feasibility.

The fitness function implemented for the edge coloring problem is the sum of two terms:

1. the objective function as given in Section 2, $P(\vec{y})$ which calculates the total number of colors used to color the edges of the graph (represented by $\vec{y}$).
2. a penalty term used to penalize infeasible strings, where the weight of the penalty is proportional to the number of violations (number of adjacent edges with the same color).

The penalty term is the sum of two parts. The first part is the maximum vertex degree to which one is added, i.e., $\Delta+1$. It is an offset term but does not guarantee that an infeasible vector will always yield a fitness value inferior to the ones obtained from feasible vectors. To have a strict separation between feasible and infeasible strings, one would have to use, for example, m: the number of edges, as an offset, since we cannot do worse than coloring m edges with m colors. But we noticed that $\Delta+1$ was good enough to be used as an offset term.

The second part of the penalty term is the number of violated pairs. This imposes a heavier penalty on infeasible strings with greater number of violated edges, thus differentiating the degree of infeasibility among strings.

The fitness function of $\vec{y} = y_1 y_2 \ldots y_m$, used with both genetic algorithms, is given by:

$$f(\vec{y}) = P(\vec{y}) + s[(\Delta + 1) + T(\vec{y})]$$

where $P(\vec{y})$ is the number of different values of the components of $\vec{y}$ (the objective function in Section 2), $s = 0$ if $\vec{y}$ is feasible and $s = 1$ otherwise, $\Delta + 1$ is the offset term, and $T(\vec{y})$ is the number of pairs of edges that violate the color condition. In other words, $T(\vec{y})$ represents the number of adjacent edges that have the same color.

4.2 Representation and Operators for the GGA

The grouping genetic algorithm, GGA [6], as the name suggests, focuses purely on problems with grouping properties. The edge coloring problem consists in placing the m edges of the graph in the minimum number of groups (colors) under certain restrictions. GGA adopts all the basic notions of the GA studied in the previous section, with an alteration in the representation of strings. To capture the grouping entity of the problem, the simple (standard) chromosome is augmented with a group part. For example, the chromosome of our hypothetical 12-edge graph given in Figure 1 is represented by the string in Figure 2. In the example, we label colors with

5	3	5	5	1	3	2	1	4	4	1	3

A	5	8	11	B	7	C	2	6	12	D	9	10	E	1	3	4

Figure 2: Chromosome in GGA representing the chromosome of Figure 1.

letters so as to be able to distinguish them from edges (that are labeled with numbers). The first half is the object part and is identical to the entire chromosome seen in the previous section, while the second half is the group part.

The fitness function of a string considers the object part of the string only and computes the fitness value as explained in the previous section.

In GGA, we use the same roulette wheel selection as we do in GA. Since the new chromosome architecture does not affect the fitness value of a string, there is no need to modify the existing selection strategy.

The crossover operator, however, has been altered to work solely with the group part. Two parents are randomly chosen from the mating pool and two crossing sites are chosen for each parent. Note that the crossing sites are chosen between groups of colors. For example, consider the group parts of two chromosomes, where the delimiters represent the crossing sites.

A	5	8	11	B	7	C	2	6	12	D	9	10	E	1	3	4

A	1	3	6	7	B	8	11	C	2	9	D	10	12	E	4	5

The entire group part between the two crossing sites of the second parent, i.e., B 8 11 C 2 9 D 10 12, is then injected into the first parent at the beginning of its crossing site.

After injecting the crossing section of the second parent into the first parent, we have:

A	5	8	11	*B*	*8*	*11*	*C*	*2*	*9*	*D*	*10*	*12*	B	7	C	2	6	12	D	9	10	E	1	3	4

Since new groups are now included, we will generally have groups with the same label. In our example, we have two groups labeled B, two labeled C and two labeled D. We resolve this redundancy by eliminating groups B, C, and D from their old membership (i.e., from the first parent string), yielding

A	5	8	11	B	8	11	C	2	9	D	10	12	E	1	3	4

The string is now free of duplicate groups; however, it is likely that it still contains duplicate objects. In this case, edges 8 and 11 are duplicates. We need to remove these objects from their old membership as well (from the first parent string), resulting in

A	5	B	8	11	C	2	9	D	10	12	E	1	3	4

This leaves out edges 6 and 7 from any group. At this stage we carry out the first–fit heuristic to reassign (recolor) these edges: it visits the groups one at a time, assigns the object to the first group that is able to accommodate it, or to a randomly selected group if none is available.

After the first offspring has been generated, the roles of the two parents are reversed and the construction of the second offspring is started. This implementation complies with the guidelines suggested by [6].

The mutation operator is quite different from a random alteration of values in the chromosomes as done in GA implementations. We adopt the strategy of probabilistically removing some groups from the chromosome and reassigning the missing objects. We consider the chromosome in Figure 2. Assume that color (group) C is removed due to probability. In doing so, we leave edges (objects) 2, 6, and 12 unassigned. Consequently, we need to perform a first–fit heuristic on these objects. Each unassigned edge is assigned to the first group that is able to take it. The strength of this mutation strategy is that, if a string is feasible before mutation, it is highly likely that it will still be feasible after mutation. In the worst case, if no other group is available, all the edges may again be assigned to the original group it initially belonged to. After crossover and mutation are performed on the group part, the object part is rearranged accordingly to reflect the changes that the group part has undergone.

5 EXPERIMENTAL RESULTS

5.1 The Test Graphs

We present the computer simulation results of the GA that uses LibGA and the GGA implementation for the edge coloring problem, and compare them with the greedy algorithm *EdgeColoringAssignment*. For our experimental runs, we use 28 simple graphs and 15 multigraphs. For the simple graphs, we consider 21 benchmark graphs from Knuth's Stanford GraphBase [12] and DIMACS [4], and seven complete graphs. From the Stanford GraphBase, we use the five book graphs (anna, david, homer, huck, jean), games120, the two miles graphs (miles250 and miles500), and nine queen graphs. From the DIMACS collection, we use the four problem instances based on the Mycielski transformation (myciel3, myciel4, myciel5 and myciel6). These problems were originally created for the vertex coloring problem. We also consider complete graphs with 20, 25, 30, 35, 40, 45, and 50 vertices, that we denote by comp20, comp25, comp30, comp35, comp40, comp45, and comp50, respectively. The number of edges of the 28 problem instances varies from 20 (myciel3) to 2596 (queen12_12).

The multigraphs we use in our experimental runs were generated by repeatedly assigning edges to randomly chosen pairs of vertices until each vertex has a fixed degree. We denote the multigraphs we generated by rexmg10_x, rexmg20_x, and rexmg30_x, where 10, 20 and 30 are the number of vertices and x is the degree and has the values of 10, 20, 30, 40 and 50.

In all three tables (Table 1, Table 2 and Table 3), we give the graph name, and the number of vertices followed by the number of edges (the chromosome size for the GAs). The density of the graph, which is the ratio of the number of edges to the number of vertices, is given, followed by the maximum vertex degree of the graph.

5.2 Results and Discussion

In our study, we use a generational genetic algorithm with roulette wheel selection, crossover, and mutation. The intial population was randomly generated. In each string the m edges were randomly assigned values in the range of 1 to m. Table 1 summarizes the results of executing the greedy algorithm and the genetic algorithms for the edge coloring problem with 21 problem instances. The results tabulated under the Greedy Algorithm are the best, the number of times the best solution was obtained (Frequency), and the mean results we obtained after 1000 runs on each of the 21 problem instances. As for the genetic algorithms, the results reported in the table were obtained by running our GA that uses LibGA once on every problem instance. For all 21 problem instances, we use the same number of generations (500), the same population size (100), the same crossover rate (0.6), and the same mutation rate (0.2).

The GGA obtained its best solution very quickly. So, we decided to run all 21 problem instances with a smaller pool size (50), smaller number of generations (100), crossover rate of 0.6, and mutation rate of 0.2. We report the results of 100 such runs of the GGA in the last columns of Table 1. The three last columns of Table 1 contain the best, the frequency the best was obtained, and the mean for all 100 runs of the GGA. Also, although we ran the genetic algorithm for 100 generations, most of the results recorded in the table were obtained before the 50^{th} and some within 15 generations. Problems that exhibit a "grouping property" are more amenable for the GGA [6]. As a matter of fact, the decision of running the LibGA-based GA only once was dictated by the fact that multiple runs did not alter the best result. We thus decided to focus on the comparison between the greedy algorithm and the GGA only for complete graphs and for multigraphs.

Table 2 summarizes the results for the seven complete graphs. The results tabulated under the Greedy Algorithm are the best, the percentage of times it found the best reported in the previous column and the mean results we obtained after 1000 runs on each of the seven problem instances. As for the GGA, we used a population size of 50, crossover rate of 0.6 and mutation rate of 0.2. The results obtained by the GGA were much better than the ones obtained by the greedy algorithm. Here too, we ran the genetic algorithm for 100 generations, but most of the results recorded in the table were obtained very early in the run, some within 15 generations. Table 2 shows that the greedy algorithm was able to achieve the success obtained by the GGA in just a few cases. For two problem instances, the results obtained by the GGA are better than the ones obtained by the greedy algorithm. For the remaining complete graphs, although the results reported under Best are similar, the frequencies for obtaining these results are far less than the GGA's.

Table 3 summarizes the results obtained from the fifteen multigraphs. The settings for both the greedy algorithm as well as the GGA are similar to the ones we described in the previous paragraph for the complete graphs. The difference between the GGA and the greedy algorithm is even more pronounced than with the complete graphs of Table 2. More precisely, in all but three, the best result obtained by the GGA is better than the ones obtained by the greedy algorithm. For the remaining three problem instances, rexmg10_20, rexmg20_10 and rexmg30_20, the results reported under Best are similar, but with a drastically lower frequency for the greedy algorithm. As far as computing time is concerned, we note that in general, the difference in running times between the greedy algorithm and the GGA increased with larger problem instances. For example, it took 4.84 seconds for the 1000 runs of the greedy algorithm to produce the 11 colors for rexmg10_10 (see Ta-

Problem Instances					Greedy Algorithm			LibGA	GGA		
Name	Vertices	Edges	Density	Δ	Best	Freq.	Mean		Best	Freq.	Mean
anna.col	138	493	0.28	71	71	100%	71	71	71	100%	71
david.col	87	406	0.21	82	82	100%	82	82	82	100%	82
homer.col	561	1629	0.34	99	99	100%	99	99	99	100%	99
huck.col	74	301	0.25	53	53	100%	53	53	53	100%	53
jean.col	80	254	0.31	36	36	100%	36	36	36	100%	36
games120.col	120	638	0.19	13	14	38.8%	14.66	14	**13**	96%	13.04
miles250.col	128	387	0.33	16	16	72.9%	16.28	16	16	100%	16
miles500.col	128	1170	0.11	38	38	85.3%	38.16	38	38	100%	38
queen5_5.col	25	160	6.40	16	16	59.7%	16.43	16	16	100%	16
queen6_6.col	36	290	8.06	19	19	11.1%	20.36	19	19	100%	19
queen7_7.col	49	476	9.71	24	24	59.8%	24.45	24	24	100%	24
queen8_8.col	64	728	11.4	27	27	8.4%	28.28	27	27	100%	27
queen8_12.col	96	1368	14.3	32	33	11.3%	34.66	34	**32**	16%	32.84
queen9_9.col	81	1056	13.0	32	32	63.5%	32.40	32	32	100%	32
queen10_10.col	100	1470	14.7	35	35	13.7%	36.18	35	35	100%	35
queen11_11.col	121	1980	16.4	40	40	66.1%	40.43	40	40	100%	40
queen12_12.col	144	2596	18.0	43	43	20%	44.09	43	43	100%	43
myciel3.col	11	20	0.55	5	5	70.4%	5.30	5	5	100%	5
myciel4.col	23	71	0.32	11	11	100%	11	11	11	100%	11
myciel5.col	47	236	0.20	23	23	100%	23	23	23	100%	23
myciel6.col	95	755	0.13	47	47	100%	47	47	47	100%	47

Table 1: Test results for the GraphBase and DIMACS problem instances.

Problem Instances					Greedy Algorithm			GGA		
Name	Vertices	Edges	Density	Δ	Best	Freq.	Mean	Best	Freq.	Mean
comp20.col	20	190	9.50	19	20	0.1%	21.987	20	55%	20.45
comp25.col	25	300	12.0	24	26	0.2%	27.856	26	96%	26.04
comp30.col	30	435	14.5	29	31	0.5%	32.724	31	93%	31.07
comp35.col	35	595	17.0	34	37	2.9%	38.554	37	100%	37.0
comp40.col	40	780	19.5	39	42	7.0%	43.302	**41**	2%	41.98
comp45.col	45	990	22.0	44	48	23.5%	49.007	**47**	19%	47.81
comp50.col	50	1225	24.5	49	52	1.1%	53.738	52	16%	52.84

Table 2: Test results for the complete graph problem instances.

ble 3), while the GGA ran for 2.34 seconds to complete the 100 generations. On the other hand, the running times for the greedy and the GGA with rexmg30_50 were 110.4 seconds and 205.68 seconds, respectively. The runs were performed on a 300 MHz PentiumII computer.

We would also like to add that no fine-tuning or tinkering with the stochastic operators was performed.

6 CONCLUSION AND FUTURE WORK

This paper investigated applications of heuristic techniques for solving the edge coloring problem. The inquiry has compared a greedy algorithm with two genetic algorithms by applying all three algorithms to GraphBase and DIMACS benchmark graphs, complete graphs, and multigraphs. Our findings support the claim that the GGA outperforms the greedy heuristic, especially for dense graphs. In future work, we intend to use a repair mechanism in place of the penalty term in the fitness function. This technique has produced promising results with the set covering problem [1]. We also intend to compare our results with those obtained by Eiben and van Hemert [5] who use an adaptive evolutionary algorithm. Another extension to this work is to use other test suites generated specifically for testing the performance of evolutionary algorithms (EAs). The EA community has created test problem generators in which random problems with certain characteristics can be generated automatically and methodically. Drawing conclusions about the behavior of an EA can be facilitated when test suites are randomly created with certain characteristics. For instance, Joe Culberson has files for the graph-coloring generators available at http://web.cs.ualberta.ca/~joe/Coloring/Generators.

ACKNOWLEDGMENTS

The authors would like to thank Thomas Erlebach from TUM for helpful discussions and careful readings of the paper. They would also like to thank one of the reviewers for comments and suggestions that improved this work.

REFERENCES

[1] Bäck, T., Schütz, M., and Khuri, S. "A comparative study of a penalty function, a repair heuristic, and stochastic operators with the set-covering problem", *Artificial Evolution: Lecture*

Problem Instances					Greedy Algorithm			GGA		
Name	Vertices	Edges	Density	Δ	Best	Freq.	Mean	Best	Freq.	Mean
rexmg10_10	10	50	5	10	11	12.5%	12.177	**10**	99%	10.01
rexmg10_20	10	100	10	20	20	0.1%	22.65	20	80%	20.2
rexmg10_30	10	150	15	30	31	2.7%	32.784	**30**	61%	30.39
rexmg10_40	10	200	20	40	41	2.5%	42.937	**40**	46%	40.54
rexmg10_50	10	250	25	50	51	1.7%	53.138	**50**	48%	50.52
rexmg20_10	20	100	5	10	11	0.6%	12.64	11	100%	11
rexmg20_20	20	200	10	20	22	3.0%	23.528	**21**	6%	21.94
rexmg20_30	20	300	15	30	32	0.4%	34.128	**31**	1%	31.99
rexmg20_40	20	400	20	40	43	10.1%	44.388	**42**	100%	42
rexmg20_50	20	500	25	50	53	6.6%	54.598	**52**	79%	52.21
rexmg30_10	30	150	5	10	12	28.3%	12.777	**11**	64%	11.36
rexmg30_20	30	300	10	20	22	0.4%	23.783	22	100%	22
rexmg30_30	30	450	15	30	33	8.5%	34.346	**32**	54%	32.46
rexmg30_40	30	600	20	40	43	1.2%	44.822	**42**	9%	42.91
rexmg30_50	30	750	25	50	53	0.8%	55.094	**52**	1%	52.99

Table 3: Test results for the multigraph problem instances.

Notes in Computer Science, vol. 1063, Springer Verlag, Berlin, pp. 320-332, 1996.

[2] Bondy J.A. and Murty, U.S.R. *Graph Theory with Applications*, McMillan, New York, 1976.

[3] Corcoran, A. and Wainwright, R. " LibGA: A user-friendly workbench for ordered-based genetic algorithm research", *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, ACM Press, pp. 111-117, 1993.

[4] DIMACS: Center for Discrete Mathematics and Theoretical Computer Science. *ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/color"*, 1990.

[5] Eiben, A. and van Hemert, J.I. "SAW-ing EAs: Adapting the fitness function for solving constrained problems", *New Methods in Optimisation, Advanced Topics in Computing*, McGraw-Hill, 1999.

[6] Falkenauer, E. "A new representation and operators for genetic algorithms applied to grouping problems," *Evolutionary Computation*, pp. 123–144. The MIT Press, Massachusetts, 1994.

[7] Fiorini, S. and Wilson, R.J. *Edge-Colourings of Graphs*, Pitman, London, 1977.

[8] Hochbaum, D., editor. *Approximation algorithms for NP-hard problems*, PWS Press, Boston, MA, 1997.

[9] Holyer, I. "The NP-completeness of edge coloring," *SIAM Journal of Computing*, vol. 10, pp. 718-720, 1980.

[10] Khuri, S., Bäck, T., and Heitkötter, J. "An evolutionary approach to combinatorial optimization problems," *Proceedings of the* 22^{nd} *ACM Computer Science Conference*, pp. 66-73, ACM Press, 1994.

[11] Khuri, S. and Chiu, T. "Heuristic algorithms for the terminal assignment problem," *Proceedings of the 1997 ACM Symposium on Applied Computing*, pp. 247-251, ACM Press, 1997.

[12] Knuth, D. *The Stanford GraphBase: A Platform for Combinatorial Computing*, ACM Press, NY, 1993.

[13] Nishizeki, T. and Kashiwagi, K. "On the 1.1 edge-coloring of multigraphs," *SIAM Journal of Discrete Mathematics*, vol. 3, pp. 391-410, 1990.

[14] Ramaswami, R. "Multiwavelength lightwave networks for computer communications," *IEEE Communications,* vol. 21, no. 2, pp. 78-88, 1993.

[15] Stinson, D. *An Introduction to the Design and Analysis of Algorithms*, The Charles Babbage Research Center, Manitoba, Canada. 2nd edition, 1987.

[16] Vizing, V. "On an estimate of the chromatic class of a p-graph" (in Russian). *Diskret. Analiz.*, vol. 3, pp. 25-30, 1964.