# A Comparison of Hybrid Evolutionary Algorithms for Graph Coloring

**Pradeep R** [1] **Raman S** [2]
[1,2] Undergraduate Students, Dept. of Computer Science & engg.,
PSG college of technology, Coimbatore – 641 004
[1] mail_pradeepr@yahoo.com [2] ramans_cse@yahoo.com

***Abstract*—Graph coloring is a problem that came about from people wanting to color adjacent countries on a map. Given a map, we would like to color countries or regions such that no adjacent country or region share the same color. The computational complexity of finding a k-coloring for a graph is NP-complete; finding its chromatic number is NP-hard. Hence we require computational models different from the classic Turing model for solving such problems. Evolutionary computing is one such model. A recent and very promising approach for combinatorial optimization is to embed local search into the framework of evolutionary algorithms. In this paper, we compare such hybrid algorithms for solving the graph-coloring problem. These algorithms combine highly specialized crossover operators and well-known local search methods namely, tabu search, simulated annealing and hill climbing. Experiments of these hybrid algorithms are carried out on large DIMACS Challenge benchmark graphs. Results tend to favor the use of the less sophisticated hill climbing over the more complex simulated annealing and tabu search as local search methods.**

***Keywords*—graph coloring hybrid evolutionary algorithms optimization**

## I. Introduction

A recent and very promising approach for combinatorial optimization is to embed local search into the framework of population based evolutionary algorithms, leading to hybrid evolutionary algorithms (HEA). Such an algorithm is essentially based on two key elements: an efficient local search (LS) operator and a highly specialized crossover operator. The basic idea consists in using the crossover operator to create new and potentially interesting configurations, which are then improved, by the LS operator. Typically, a HEA begins with a set of configurations (population) and then repeats an iterative process for a fixed number of times (generations). At each generation, two configurations are selected to serve as parents. The crossover operator is applied to the parents to generate a new configuration (offspring). Then the LS operator is used to improve the offspring before inserting the latter into the population. To develop a HEA for an optimization problem, both the LS and crossover operators must be designed carefully. Nevertheless, it is usually more difficult to develop a meaningful crossover operator than a LS operator. Indeed, while various good LS algorithms are available for many well-known optimization problems, little is known concerning the design of a good crossover. In general, designing a crossover requires first the identification of some 'good properties' of the problem that must be transmitted from parents to offspring and then the development of an appropriate recombination mechanism. Note that random crossovers used in standard genetic algorithms are meaningless for optimization problems.

HEAs have recently been applied to some well-known NP-hard combinatorial problems such as the traveling salesman problem, the quadratic assignment. The results obtained with these HEAs are quite impressive. Indeed, they compete favorably with the best algorithms for these problems on well-known benchmarks. These results constitute a very strong indicator that HEAs are among the most powerful paradigms for hard combinatorial optimization.

The problem of graph coloring was made famous by the *4-color conjecture: Every map can be colored using at most four colors, such that neighboring countries do not have the same color.* Graph Coloring has a wide range of applications such as radio channel frequency assignment, register allocation, bus allocation in VLSI design, traffic control and generally resource allocation problems. The actual decision problem known as **k-coloring** is to find if the graph can be colored using at most 'k' colors. The related optimization problem is to find the smallest such number 'k', called the **chromatic number** of the graph. These problems become intractable when solved using the Turing machine model. Hence we need to explore other models of computations to find solutions to these problems.

In this paper, we are concerned with comparing the performances of different local search methods, when used in the HEA framework We analyze the performances of these methods on standard DIMACS II challenge benchmark graphs. Of these methods, the performances of the HEA using tabu search (Galiner & Hao, 1999) and the HEA using simulated annealing (Fotakis et al) have been studied.

We are also concerned with the applicability of such HEAs to graph coloring in on line applications, to

find an optimal coloring rather than checking if it is k-colorable. In online applications, the time taken for finding the (near-)optimal coloring is also important.

We have arranged the text as follows: section II reviews the existing methods for graph coloring. Section III & IV build the preliminaries. Section V describes the implementation. Sections VI to VIII report experimental data. Section IX concludes and Section X lists out the references.

## II. Existing Heuristic Methods

Let $G = (V, E)$ be an undirected graph with a vertex set $V$ and an edge set $E$. A subset of $G$ is called an independent set if no two adjacent vertices belong to it. A k-coloring of $G$ is a partition of $V$ into $k$ independent sets (called proper color classes). An optimal coloring of $G$ is a k-coloring with the smallest possible $k$ (the chromatic number of $G$). The graph coloring problem is to find an optimal coloring for a given graph $G$. Many heuristics have been proposed for this problem. We review below the main classes of known heuristics.

*Greedy constructive methods*: The principle of a greedy constructive method is to color successively the vertices of the graph. At each step, a new vertex and a color for it are chosen according to some particular criteria. These methods are generally very fast but produce poor results. Two well-known techniques are the saturation algorithm and the Recursive Largest First algorithm (Leighton, 1979).

*Genetic algorithm (GA)*: The main principles of GAs are presented in Holland (1975), Goldberg (1989) [5]. Davis proposed to code a coloring as an ordering of vertices. The ordering is decoded (transformed into a coloring) by using a greedy algorithm. This algorithm gives poor results. No other work using pure GA has been reported since then. It is believed that pure GAs are not competitive for the problem. The recently introduced a set of genetic operators for the so-called "grouping problems". These operators are applicable to the GCP, which is a special grouping problem, but no results are available on benchmark graphs.

*Local search (LS)*: Several LS algorithms have been proposed for the GCP and prove very successful. These algorithms are usually based on simulated annealing or tabu search. They differ mainly by their way of defining the search space, the cost function and the neighborhood function. Hertz and de Werra proposed a penalty approach and a simple neighborhood. They tested simulated annealing and tabu search

*Hybrid algorithms*: Hybrid algorithms integrating local search and crossovers were recently proposed in Fleurent and Ferland (1996) [1], Costa et al. (1995). Fleurent and Ferland (1996) shows that crossovers can improve on the performance of LS, but this happens only for a few graphs and needs important computational efforts. Hybrid algorithms using a population and local search but without crossover have also been proposed, producing excellent results on benchmark graphs.

*Successive building of color classes*: This particular strategy consists in building successively different color classes by identifying each time a maximal independent set and removing its vertices from the graph. So the graph-coloring problem is reduced to the maximal independent set problem. Different techniques to find maximal independent sets have been proposed for building successive color classes, including local search methods (Fleurent and Ferland, 1996 [1]). This strategy proves to be one of the most effective approaches for coloring large random graphs.

## III. Crossovers for Graph Coloring

Standard genetic algorithms were initially defined as a general method based on blind genetic operators (crossover, mutation, etc.,) able to solve any problem whose search space is coded by bit strings. But it is now admitted that the standard GA is generally poor for solving optimization problems and that genetic operators, notably crossover, must incorporate specific domain knowledge. More precisely, designing crossovers requires to identify properties that are meaningful for the problem and then to develop a recombination mechanism in such a way that these properties are transmitted from parents to offspring.

Concerning graph coloring, there are essentially two different approaches according to whether a coloring is considered to be an assignment of colors to vertices or to be a partition of vertices into color classes.

The *assignment approach* considers a configuration (a proper or improper) coloring as an assignment of colors to vertices. So, it is natural to present a configuration $s$ by a vector $\{s(v_1), s(v_2), \ldots, s(v_n)\}$ of size $|V|$, each $s(v_i)$ being the color assigned to the vertex $v_i$. It is now possible to define an uniform assignment crossover that assigns to a vertex in the child the color class associated with the same vertex in either parent, with an equal probability 0.5.

Assignment crossovers have been proposed for coloring problem in Fleurent and Ferland (1996) [1], Costa et al. (1995). In Fleurent and Ferland (1996), the uniform assignment crossover is enhanced in the following way: if a vertex $v$ is conflicting in one of the two parents, it systematically receives the color assigned to $v$ in the other parent. We can observe that with the assignment approach, the property transmitted by crossovers is a couple (vertex, color): a particular vertex re-

ceives a particular color. Nevertheless, such a couple, considered in isolation, is meaningless for graph coloring since all the colors play exactly the same role.
For the coloring problem, it will be more appropriate and significant to consider a *pair* or a *set of vertices* belonging to a same class. This leads to a different approach, called the *partition approach*. With this approach, a configuration is considered as a partition of vertices into color classes and a crossover is required to transmit color classes or subsets of color classes. Given this general principle, many different crossovers are now imaginable. One possibility is to build first a partial configuration of maximum size from subclasses of the color classes of the two parents and then to complete it with an appropriate heuristic with the remaining unassigned vertices, to obtain a complete configuration.

One way to construct the partial configuration is to build each color class successively in a greedy fashion: both parents are considered successively and we choose in the considered parent the class with the maximal number of unassigned vertices to become the next class. Note that this crossover, described in Galiner & Hao (1999) as Greedy Partition Crossover (GPX), generates each time one offspring. The GPX crossover is presented in detail in Section IV.4.

Finally, Galiner & Hao have carried out experiments to confirm the pertinence of the characteristic considered by partition crossovers (i.e. set of vertices). For a random graph and a fixed $k$ sample k-colorings are collected. Analysis of the frequency with which two non-adjacent vertices are grouped into a same color discloses that some pairs of non-adjacent vertices are much more frequently grouped into a same class than others. This constitutes a positive indicator of the applicability of the partition approach.

## IV. The Hybrid Evolutionary Algorithm

To solve the GCP, i.e. to color a graph with a minimum possible number of colors, a particular approach consists in applying a coloring algorithm to look for a k-coloring for a (sufficiently high) number of colors $k=k_0$. Then whenever a k-coloring is found, one re-applies the same algorithm to look for k-colorings with decreasing numbers of colors ($k= k_0-1, k_0-2, \ldots$) Therefore, the graph coloring problem is reduced to solving increasingly difficult k-coloring problems. This might prove costly in terms of time especially for online applications of the method. It is efficient only when the chromatic number is known approximately. Later in section V, we explore certain methods to cut down on the number of values of $k$ that need to be tested. In this section, the components of our Hybrid Coloring Algorithm (HCA) for finding k-colorings are presented.

### *1. Search space and cost function*

The k-coloring problem consists in coloring a graph $G=(V,E)$ with a fixed number of $k$ colors. To solve a k-coloring problem, we consider the set of all possible partitions of $V$ in $k$ classes including those that are not proper k-colorings. Each partition is attributed a penalty equal to the number of edges having both endpoints in a same class. Therefore, a partition having a 0 penalty corresponds to a proper k-coloring (a solution). The purpose of the search algorithm is then to find a partition having a penalty as small as possible. So solving the minimization problem can solve the k-coloring problem. In the following we call configuration any element of the search space $S$ and reserve the word solution for a proper k-coloring.

### *2. The general procedure*

In HCA, the population P is a set of configurations having a fixed constant size $|P|$. We present the general algorithm below.The algorithm first builds an initial population of configurations (*InitPopulation*) and then performs a series of cycles called *generations*. At each generation, two configurations $s1$ and $s2$ are chosen in the population (*ChooseParents*). A crossover is then used to produce an offspring $s$ from $s1$ and $s2$ (*Crossover*). The LS operator is applied to improve $s$ for a fixed number $L$ of iterations (*LocalSearch*). Finally, the improved configuration $s$ is inserted in the population by replacing another configuration (*UpdatePopulation*). This process repeats until a stop condition is verified, usually when a pre-fixed number *MaxIter* of iterations is reached. Note however that the algorithm may stop before reaching *MaxIter* , if the population diversity becomes too small.

**The hybrid coloring algorithm**
**Data :** *graph G =(V, E), integer k*
**Result :** *the best configuration found*
**Begin**
  P=*InitPopulation(|P|)*
  **While** *not Stop-Condition ()* **do**
    (s1,s2)=*ChooseParents*(P)
    s=*Crossover*(s1,s2)
    s=*LocalSearch*(s, *parameters*)
    P=*UpdatePopulation*(P,s)
**End**

It is noticed that this hybrid algorithm differs from a genetic algorithm by some features. The fundamental difference is naturally that the random mutation operator of a GA is replaced with a LS operator. Another difference concerns the selection operator of a GA, which encourages the survival and reproduction of the best individuals in the population. In the hybrid algorithm, the selection is ensured jointly by *ChooseParents* and *UpdatePopulation*.

### *3. The initialization operator*

The operator *InitPopulation(|P|)* initiates the population $P$ with $|P|$ configurations. To create a configuration, we use the greedy saturation algorithm of Br´elaz (1979) slightly adapted in order to produce a partition of $k$ classes (adapted from Galiner & Hao [3]). The algorithm works as follows. We start with $k$ empty color classes $V_1 \ldots V_k$. At each step, we chose a vertex $v \in V$ such that $v$ has the minimal number of allowed classes (i.e. a class that does not contain any vertex adjacent to $v$). To put $v$ in a color class, we chose among all the allowed classes of $v$ the one $V_i$ that has the minimal index $i$. In general, this process cannot assign all the vertices. Each of unassigned vertex is then put into one color class randomly chosen. Once a configuration is created, it is immediately improved by the LS operator for $L$ iterations. Due to the randomness of the greedy algorithm and the LS improvement, the configurations in the initial population are quite different. This point is important for population based algorithms because a homogeneous population cannot evolve efficiently.

*4. The crossover operator*

The crossover used here is the GPX crossover presented. Let us show now how this crossover works. Given two parent configurations $s1 = \{V_1^1, \ldots, V_k^1\}$ *and* $s2 = \{V_1^2, \ldots, V_k^2\}$ chosen randomly by the *ChooseParent* operator from the population, the algorithm *Crossover(s1,s2)* builds an offspring $s = \{V_1,\ldots,V_k\}$ as follows:

**The GPX crossover algorithm**

**Data :** *configurations* $s1 = \{ V_1^1, \ldots, V_k^1\}$ *and* $s2 = \{ V_1^2, \ldots, V_k^2\}$

**Result :** *configuration* $s = \{V_1,\ldots,V_k\}$

**Begin**

**For** $l(1 <= l <= k)$ **do**

1. if $l$ is odd, then $A := 1$, else $A := 2$
2. choose $i$ such that $V_i^A$ has a maximum cardinality
3. $V_l := V_i^A$
4. Remove the vertices of $V_l$ from $s1$ and $s2$

**End**

Assign randomly vertices $V - ( V_1 \cup V_2 \cup \ldots \cup V_K)$

**End**

The algorithm builds step by step the $k$ classes $V_1,\ldots,V_K$ of the offspring at step $l(1 <= l <= k)$, the algorithm builds the class $V_l$ in the following way. Consider parent $s1(A = 1)$ or parent $s2(A = 2)$ according to whether $l$ is odd or even. In the considered parent, choose the class having the maximum number of vertices to become class $V_l$ and remove these vertices from parents $s1$ and $s2$. At the end of these $k$ steps, some vertices may remain unassigned. These vertices are then assigned to a class randomly chosen. Let us notice that the edges of the graph, which are used in the cost function f, are not involved in the crossover operator.

*5 The LS operators*

*5.1 Tabu Search*

The purpose of the LS operator *LocalSearch(s, L)* is to improve a configuration $s$ produced by the crossover for a maximum of $L$ iterations before inserting $s$ into the population. In general, any local search method may be used. In our case, we use tabu search (TS), an advanced local search meta-heuristic (Glover and Laguna, 1997). TS performs guided search with the help of short and eventually long term memories. Like any LS method, TS needs a neighborhood function defined over the search space $N:S \rightarrow 2^S$. Starting with a configuration, a typical TS procedure proceeds iteratively to visit a series of locally best configurations following the neighborhood. At each iteration, a *best* neighbor is chosen to replace the current configuration, even if the former does not improve the current one. This iterative process may suffer from cycling and get trapped in local optima. To avoid the problem, TS introduces the notion of *Tabu lists*. The basic idea is to record each visited configuration, or generally its attributes and to forbid re-visiting this configuration during next $tl$ iterations ($tl$ is called the tabu tenure).

The TS algorithm used here is the one suggested by Galiner & Hao (1999) [3] which is actually an improved version of the TS algorithm proposed by Hertz and de Werra (1987). Here a *neighbor* of a given configuration $s$ is obtained by moving a single vertex $v$ from a color class to another color class $V_i$. To make the search more efficient, the algorithm uses a simple heuristic: the vertex $v$ to be moved must be conflicting with at least another vertex in its original class. Thus a neighboring configuration of $s$ is characterized by a *move* defined by the couple $(v, i) \in V \times \{1 \ldots k\}$. When such a move $(v, i)$ is performed, the couple $(v, s(v))$ is classified tabu for the next $tl$ iterations, where $s(v)$ represents the color class of vertex $v$ in $s$. Therefore, $v$ cannot be reassigned to the class $s(v)$ during this period. Nevertheless, a tabu move leading to a configuration better than the best configuration found so far is always accepted (*aspiration criterion*). The tabu tenure $tl$ for a move is variable and depends on the number $nb_{CFL}$ of conflicting vertices in the current configuration: $tl = Random(A) + \alpha \times nb_{CFL}$ where $A$ and $\alpha$ are two parameters and the function $Random(A)$ returns randomly a number in [0,A]. To implement the tabu list, it is sufficient to use a $V \times \{1 \ldots k\}$ table.

The algorithm memorizes and returns the *most recent* configuration $s^*$ among the best configurations found: after each iteration, the current configuration $s$ replaces $s^*$ if $f(s) <= f(s^*)$ (and not only if $f(s) < f(s^*)$). The rational to return the last best configuration is that

we want to produce a solution which is as far away as possible from the initial solution in order to better preserve the diversity in the population. The skeleton of the TS algorithm is given below.

**The TS operator**
**Data :** *graph G = (V, E)*, configuration *s*0
**Result :** the best configuration found
**Begin**
*s* := *s*0
**While** *not Stop-Condition()* **do**
choose a best authorized move *(v,i)*
introduce the couple *(v,s(v))* in the Tabu list for *tl* iterations
perform the move *(v,i)* in *s*
**End**

The configuration created by a crossover and improved by TS is now to be inserted in the population. To do this, the worst of the two parents is replaced.

*5.2 Simulated Annealing*

The basic idea behind simulated annealing is that the search process can escape local minima if we allow some uphill moves. Starting with an initial solution $s = s0$ we choose at random a neighbor of $s$. If that move improves the cost function we perform it; otherwise we move to that solution with probability $p = exp(-\Delta/t)$, where $\Delta$ is the deterioration in the cost function. The parameter $t$, called *temperature*, is used in order to control the acceptance ratio: In the beginning, the temperature is set to a high value, and therefore a move to a worse is solution is often; as the search goes on, the temperature is reduced by a factor $a$ and thus the probability of a downhill move decreases. The initial temperature together with the factor $a$ and the number of trials performed in each iteration is called the *cooling schedule*. The procedure that follows implements a full annealing cycle.

**The simulated annealing operator**
**Data:** $s0$: initial configuration, $f()$: cost function, $N(s)$: the neighbors of $s$,
**Result:** *improved configuration:* $s0$
Compute an initial temperature $t0$ depending on $f(s0)$
Let $t0 = tf * t0 + tp$ where *tf, tp* are user-defined parameters
**Repeat**
**Repeat**
Compute the size of the neighborhood $N = |N(s0)|$
Let $nrep = N * Sf$, where *Sf* is a user-defined parameter
Randomly choose $s \in N(s0)$
$\Delta = f(s) - f(s0)$
**If** $\Delta < 0$
Let $s0 = s$
**Else**
With probability $exp(-\Delta/t0)$ set $s0 = s$
**Until** iterations = nrep
Let $t0 = a * t0$
**Until** term cond SA = TRUE

This algorithm is adapted from the HEA described by D.A. Fotakis et al. Here the neighborhood size is re-computed for every solution and therefore varies from time to time (the neighborhood size is analogous to the $nb_{CFL}$ parameter for tabu search). This feature makes a difference if we choose a neighborhood that reduces its size over the optimization process. As we shall see in the next section, that is the case in our implementation for the GCP. The parameters that need to be defined by the user for the *anneal* procedure are namely *tf, tp, sf, a*. These parameters are used so that we shall be able to control better the cooling schedule of the annealing process. It is well known that maintaining the diversity of the population in high levels is very important in GAs, because otherwise the crossover is not able to produce different configurations and that leads to premature convergence. This is even more important in HEAs as they usually use small populations (5-10). On one hand, SA tends to maintain diversity by exploring solutions when the temperature is high. On the other hand, we can see annealing as a greedy local search (using zero temperatures), which could be handy when we are dealing with good solutions that we want to improve.

We also use a simulated annealing operator that uses a static cooling schedule with a fixed starting temperature and fixed number of iterations. We refer to the previously described dynamic neighborhood based simulated annealing operator as DSA and the simulated annealing operator with a static cooling schedule as SA

*5.3 Hill Climbing*

The greedy local search that becomes of simulated annealing at zero temperature is referred to as hill climbing. It differs from simulated annealing, because it never accepts a configuration worse than the current configuration. It is the simplest of the local search strategies.

## V. Finding the optimal coloring

The general way to solve the optimization problem using solutions for the decision problem is to proceed in a linear fashion. Here for finding the minimal coloring, we first start by trying to find a *k*-coloring and

if that is successful, we proceed to find a (*k-1)*-coloring, (*k-2)*-coloring and so on, until we find *n* such that we cannot perform a (*k-n)*-coloring for that particular graph. Then the minimum number of colors needed to color the graph is *k-n+1*. This approach might suffice for offline applications and cases where the chromatic number is estimated (approximately) already. However for online applications, this leads to a serious inefficiency.

We use a method, analogous to binary search and the bisection method of root finding that can find an optimal coloring for the graph by trying at most $log_2|V|$ *k*-coloring instances for a graph G=(V,E).

**The Binary Minimal Color Search Algorithm**
**Data:** Graph *G = (V,E), j=0, l=|V|*
**Result:** The minimal coloring for G in *l*
**Do**
Let *k=(j+l)/2*
**If** G is *k*-colorable, *l = k*
**Else** *j = k*
**Until** (*l-j > 2*)

By setting a timeout *t* for each attempt at a *k*-coloring, we can be sure that the attempt at finding the minimal coloring does not take time more than $t\ log_2|V|$. This is an important consideration for online applications of graph coloring.

## VI. Results

The two factors that are considered while testing the effectiveness of different local search heuristics for graph coloring are (1) The minimal coloring obtained (2) The total time taken to obtain the minimal coloring. The graphs were tested using several DIMACS Challenge Benchmarks and some real world application graphs.

There remains a word to be said about the performance of the simulated annealing operators, before we proceed to discuss the results. The simulated annealing operator with a dynamic cooling schedule (DSA) fails to perform on most graphs and obtains only mediocre results. The minimal coloring obtained with DSA is in most cases far behind that of the results obtained with other local search methods. On the other hand the simulated annealing operator with a static cooling schedule, performs exceptionally well for certain values of starting temperature. Hence we have omitted data pertaining to the DSA operator in our results.

The results are tabulated as two tables: table1 shows the median (mode, where applicable) of the minimum colorings obtained on different graphs; table2 shows the best minimal coloring obtained for each method on various graphs.

In the tables (1 & 2), we have shown the best values among different local search methods themselves in a bold font and the second best values in a bold italic font. We observe that the best values obtained are mostly with SA1. However almost all results obtained with HC are either best or second best. In table2, HC obtains a major share of the best results themselves, compared with other operators. The following table (table3) gives the average time taken for finding the minimal coloring for different graphs, weighted by the number of vertices. Here also we see that the HC operator dominates. Considering both performance and time, we find that the HC operator is efficient.

The results are also shown graphically as plots (fig1 and fig2). The first plot shows how the profile of the curve corresponding to the HC operator closely matches the profile of the simulated annealing operator (SA1), which is the best operator in terms of the median value of the minimal coloring(figure1). Similarly the best values produced by HC also closely match that of those produced by TB, the tabu search operator, the best operator in finding the lowest values for minimal coloring (figure2).

Also HC is better than SA1 or TB in terms of the time taken to find solutions. All these suggest that Hill Climbing is better suited for online applications.

Results were obtained on a Pentium III 450 Mhz computer running Windows 98. The algorithms were implemented in C++ and compiled using an optimizing C++ compiler (g++ with maximum level of optimization -O3).

## VII. Tables of Experimental Data

| | TS | SA1 | SA2 | SA3 | HC |
|---|---|---|---|---|---|
| DSJC250.9 | 102 | 91 | 101 | 101 | *99* |
| DSJC250.5 | *39* | 37 | 37 | 42 | 40 |
| DSJC250.1 | *12* | 13 | 13 | 10 | *12* |
| DSJC125.9 | 56 | *53* | 54 | 52 | 54 |
| DSJC125.5 | *22* | 24 | 21 | 24 | *22* |
| DSJC125.1 | *7* | *7* | *7* | *7* | *7* |
| flat300_26_0 | 45 | 44 | 50 | 37 | *38* |
| flat300_28_0 | 44 | 49 | 48 | 51 | *46* |
| flat300_20_0 | 46 | 32 | 40 | *36* | *36* |
| mulsol.i.1 | *50* | 49 | *50* | *50* | 49 |
| mulsol.i.2 | 31 | 31 | *32* | 31 | *32* |
| mulsol.i.3 | 31 | 31 | 31 | 31 | 31 |
| mulsol.i.4 | 31 | *32* | 31 | 31 | 31 |
| mulsol.i.5 | 31 | 31 | 31 | 31 | 31 |
| zeroin.i.1 | 49 | *50* | *50* | 49 | 49 |
| zeroin.i.2 | 30 | 30 | *31* | 30 | 30 |
| zeroin.i.3 | 30 | 30 | 30 | 30 | *31* |

Table1 – *tabulation of median/mode of the Minimal colorings obtained by various local search operators on various test/application graphs.*

| | TS | SA1 | SA2 | SA3 | HC |
|---|---|---|---|---|---|
| DSJC250.9 | 91 | *87* | *87* | 85 | 89 |
| DSJC250.5 | 35 | *34* | 35 | 39 | 33 |
| DSJC250.1 | *12* | *12* | *12* | 9 | *12* |
| DSJC125.9 | 54 | 45 | 53 | *46* | *46* |
| DSJC125.5 | 19 | *22* | 19 | 19 | 19 |
| DSJC125.1 | 6 | 6 | *7* | *7* | 6 |
| flat300_26_0 | 40 | 39 | 44 | *37* | 36 |
| flat300_28_0 | 39 | 49 | 43 | *38* | 37 |
| flat300_20_0 | 39 | *27* | *27* | 30 | 26 |
| mulsol.i.1 | 48 | *49* | *49* | *49* | *49* |
| mulsol.i.2 | 31 | 31 | 31 | 31 | 31 |
| mulsol.i.3 | 31 | 31 | 31 | 31 | 31 |
| mulsol.i.4 | 31 | 31 | 31 | 31 | 31 |
| mulsol.i.5 | 31 | 31 | 31 | 31 | 31 |
| zeroin.i.1 | 49 | 49 | 49 | 49 | 49 |
| zeroin.i.2 | 30 | 30 | 30 | 30 | 30 |
| zeroin.i.3 | 30 | 30 | 30 | 30 | 30 |

Table2 – *tabulation of the best minimal colorings obtained by various local search operators obtained on various test/application graphs.*

** TS – tabu search operator*
** SA1, SA2, SA3 – Simulated Annealing operators with (initial temperature, alpha) = (1000,0.8),(10000,0.8)&(100,0.8) respectively,*
** HC – Hill Climbing Operator*

| | TS | SA1 | SA2 | SA3 | HC |
|---|---|---|---|---|---|
| Average time (s) | 68.57 | 62.61 | 39.39 | 36.94 | 44.46 |

Table3 – *Average time taken (in seconds) by various local search operators.*

Note: The DIMACS Challenge benchmark graphs are available via ftp from ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/color/

## VIII. Graphical Plots of Experimental Data

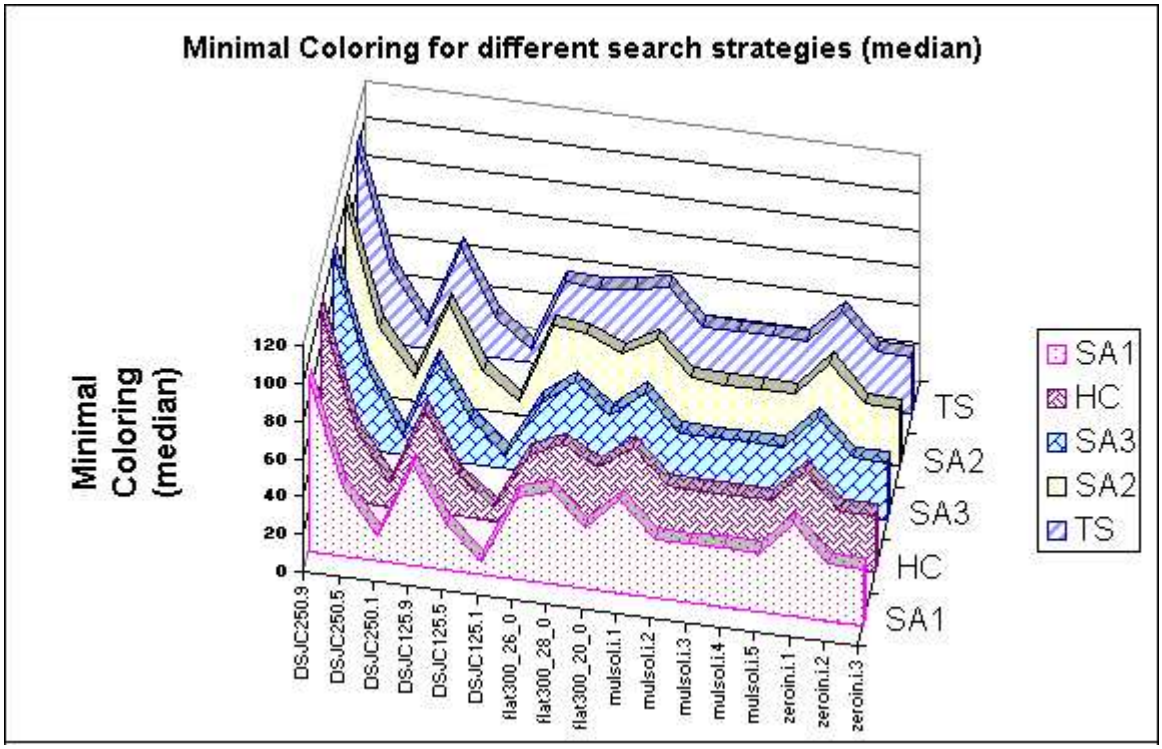


Figure 1 – *plot of median values of the minimal colorings obtained by different local search operators*

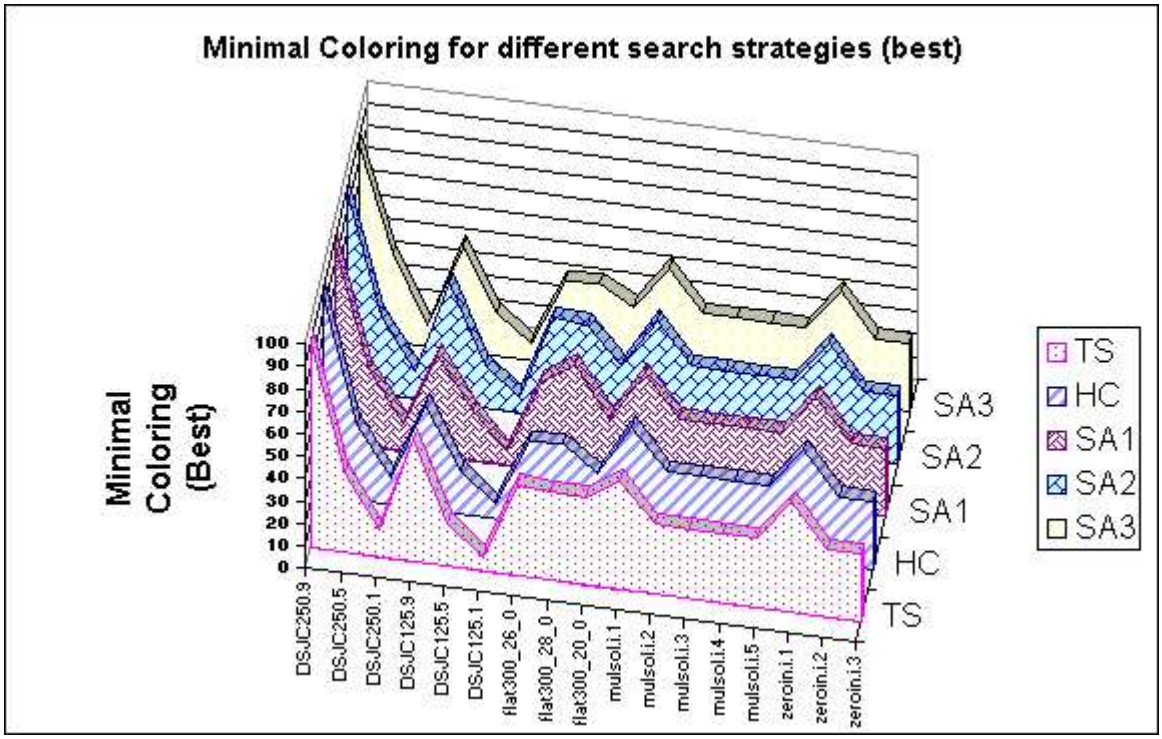


Figure 2 – *plot of best values of the minimal colorings obtained by different local search operators*

## IX. Conclusion

As a conclusion, the simulated annealing operator (SA1) is consistent, while the tabu search algorithm is effective. Hill climbing is quite fast as expected, due to its simplicity but is also quite consistent and effective too. In short, Hill climbing has all the desirable properties and is a good choice as LS operators in HEAs for graph coloring problems, especially for online applications.

## X. References


1. Fleurent C. and Ferland J.A. – *Object-Oriented Implementation of Heuristic Search Methods for Graph Colorin, Maximum Clique and Satisfiability,* in proceedings of the 2nd DIMACS Implementation challenge, DIMACS series in Discrete Mathematics and Theoritical Computer Science, D.S.Johnson and M.A.Trick (Eds.), American Mathematical Society, vol. 26, 1996, pp. 619-652

2. Fotakis, Dimitris A. , Likothanassis, Spiridon D. and Stefanakos Stamatis K. - *An Evolutionary Annealing Approach to Graph Coloring*

3. Galiner, Philippe and Hao, Jin-Kao – *Hybrid Evolutionary Algorithms for graph coloring, Journal of Combinatorial Optimization 3, 379–397 (1999)*

4. Goldberg, David E., - *Genetic Algorithms in Search, Optimization and Machine Learning* (Addison-Wesley, 1989)

5. Glover F. and Laguna M., *Tabu Search*, Kluwer Academic Publishers, 1997.
   Goldberg, David E., *Genetic Algorithms in Search; Optimization and Machine Learning*,
   Addison-Wesley, 1989.

6. Kirkpatrick S., Gelatt C.D. Jr., Vecchi M.P. - *Optimization by Simulated Annealing*, Science, Number 4598, 13 May 1983