Journal of Heuristics, 4: 25–46 (1998)

Graph Coloring with Adaptive Evolutionary Algorithms

A.E. EIBEN, J.K. VAN DER HAUW AND J.I. VAN HEMERT
Leiden University

Abstract

This paper presents the results of an experimental investigation on solving graph coloring problems with Evolutionary Algorithms (EAs). After testing different algorithm variants we conclude that the best option is an asexual EA using order-based representation and an adaptation mechanism that periodically changes the fitness function during the evolution. This adaptive EA is general, using no domain specific knowledge, except, of course, from the decoder (fitness function). We compare this adaptive EA to a powerful traditional graph coloring technique DSatur and the Grouping Genetic Algorithm (GGA) on a wide range of problem instances with different size, topology and edge density. The results show that the adaptive EA is superior to the Grouping (GA) and outperforms DSatur on the hardest problem instances. Furthermore, it scales up better with the problem size than the other two algorithms and indicates a linear computational complexity.

Key Words: evolutionary algorithms, genetic algorithms, constraint satisfaction, graph coloring, grouping problem, penalty functions, adaptive parameters

1. Introduction

The main goal of this paper is to present the results of an experimental study on solving graph coloring problems by Evolutionary Algorithms (EAs). In particular we show the working of a new, problem independent adaptive mechanism for constraint handling in EAs. In Section 2 a brief overview of graph coloring problems and graph coloring techniques is given. We decide to investigate graph instances that are 3-colorable and identify the instances around the so-called phase transition as the most challenging ones. Two graph coloring algorithms are selected to serve as competitors to our EAs. Among the traditional graph coloring techniques we choose DSatur from Brélaz for its reported high performance (Brélaz (1979)). Besides, we also test the Grouping GA of Falkenauer, because graph coloring can be seen as a grouping problem and the GGA shows excellent performance on grouping problems, such as bin packing (Falkenauer (1994)). In Section 3 performance measures for comparison of different algorithms are discussed. Thereafter, in Section 4 the Grouping GA is described; in Section 5 genetic algorithms with integer representation and with order-based representation are compared. In Section 6 we present our adaptive mechanism that is modifying the penalty function during the evolution, and we show that this adaptive mechanism highly increases performance. In Section 7 a big comparison between our adaptive EA, the Grouping GA and DSatur is made. We observe that the Grouping GA exhibits the lowest performance and that the adaptive EA outperforms DSatur on the hardest problem instances; moreover, it scales up better with the problem size. Finally, we draw conclusions in Section 8.

2. Graph coloring

The main motivation of the present research is our interest in the applicability of genetic, or more generally evolutionary, algorithms to constraint satisfaction problems. In general, a *constraint satisfaction problem* (CSP) is a pair $\langle S, \phi\rangle$, where S is a free search space, i.e., a Cartesian product of sets $S = D_1 \times \cdots \times D_n$, and ϕ is a formula, a Boolean function on S (Eiben and Ruttkay (1997)). A *solution of a constraint satisfaction problem* is an $s \in S$ with $\phi(s) = true$. Usually a CSP is stated as a problem of finding an instantiation of variables $v_1, \ldots, v_n$ within the finite domains $D_1, \ldots, D_n$ such that constraints (relations) $c_1, \ldots, c_m$ prescribed for (some of the) variables hold. The formula ϕ is then the conjunction of the given constraints. It has been proved that every CSP can be equivalently transformed to a binary one, i.e., to a CSP where each constraint concerns exactly two variables (Nudel (1983)). For graph coloring problems this property is natural, if we envision nodes as variables and edges as constraints. This suggests that the findings on using a genetic algorithm with no specific knowledge on the problem structure can be applicable to other CSPs as well.

In the family of graph coloring problems an undirected graph $G = (V, E)$ is given and the problem is to color each node in such a way that no two nodes connected by an edge are colored with the same color. There exist several variations of this problem, like finding the least number of colors that is needed to color the graph, or to find the largest subgraph in G that can be colored with the given number of colors. All of these problems are known to be NP-complete (Garey and Johnson (1979)), so it is unlikely that a polynomial-time algorithm exists that solves any of these problems (Arora et al. (1992)). A pure constraint satisfaction problem is the *graph 3-coloring* variant, where each node in the graph is to be colored with one of three given colors. In this problem variant, there is no optimization involved, such as minimizing the number of colors used for a conflict-free coloring. We have chosen to perform an in-depth investigation of this variant, restricting ourselves to one problem type, but studying three different graph topologies, several problem sizes and a whole range of edge connectivity values, yet keeping the number of experiments manageable.

2.1. Problem instances

In the literature there are not many benchmark 3-colorable graphs and therefore we create graphs to be tested with the graph generator[1] written by Joe Culberson. This generator creates various classes of k-colorable quasi-random graphs. The classes we want to investigate are the following. *Arbitrary 3-colorable graphs* where vertices are randomly assigned to one of the three partition elements (colors) uniformly and independently. This is a class that has widely been investigated. We use the term ‘arbitrary’ to distinguish these kind of graphs from the following ones. *Equi-partite 3-colorable graphs* where the three color sets are as nearly equal in size as possible (the smallest sets having one element less than the largest). Culberson reports that these graphs present the most difficulty in obtaining the specified coloring for a suite of various algorithms (Culberson and Luo (1996)). Because the sets are almost equal in size, an algorithm has less information to make use of. In case of *flat 3-colorable graphs* also the variation in degree for each node is kept to a minimum, so this class is even tougher, since even less information is available to the algorithm. Creating test graphs happens by first pre-partitioning the vertices in three sets (3 colors) and then drawing edges randomly. For the first two types of graphs, once the pre-partitioning has been done, a vertex pair v, w is assigned an edge with fixed probability p, provided that v and w are not in the same color set. So there is some variation in the degree for each vertex. This measure p is called the *edge connectivity* of a graph instance. When creating the test graphs a random number generator is needed. This generator is fed with a random seed, which obviously influences the created graphs, and—as it turns out from the experiments—also effects the performance of the graph coloring algorithms.

In our preliminary experiments comparing different genetic algorithms equi-partite graphs will be used (Sections 5 and 6), while using all three classes in a final comparison with two other methods (Section 7). For specifying the investigated graph instances we use the notation $G_{eq,n=200,p=0.08,s=5}$, standing for an equi-partite 3-colorable graph with 200 vertices, edge probability 8% and seed 5 for the random generator.

It is known that for many NP-complete problems, typical instances can be very easy to solve. Turner found that many k-colorable graphs are easy to color, so comparisons between algorithms based on these graphs are not very meaningful (Turner (1988)). For an interesting comparison one should look for hard problem instances that pose some challenges for candidate algorithms. Cheeseman et al. (1991) found that NP-complete problems have an ‘order parameter’ and that the hard problems occur at a critical value or *phase transition* of such a parameter. For graph coloring, this order parameter is the edge probability or edge connectivity p. Using the cost function estimation of Clearwater and Hogg (1996), one can determine the approximate location of the phase transition depending on the number of nodes n, in the graph. The phase transition for our type of problems will occur when the edge connectivity values are around $7/n$–$8/n$. These values are a bit different from the estimate in (Cheeseman, Kenefsky, and Taylor (1991)), but our experiments confirm this range and indicate that the hardest graphs are those with an edge connectivity around $7/n$–$8/n$, independently from the applied graph coloring algorithm. In this investigation we concentrate on graphs in this region.

2.2. Graph coloring algorithms

There are many traditional graph k-coloring techniques, based on heuristics. Some existing algorithms are: an $O(n^{0.4})$-approximation algorithm by Blum (1989), the simple Greedy algorithm (Kučera (1991)), DSatur from (Brélaz (1979)), Iterated Greedy (IG) from (Culberson and Luo (1996)), XRLF from (Johnson et al. (1991)). Probably the most simple and best known algorithm is the Greedy algorithm, which takes some ordering of the nodes of a graph and colors the nodes in this order with the smallest color[2] that is possible without violating constraints. Grimmet and McDiarmid (1975) have shown that for almost all random graphs the Greedy algorithm uses no more than about twice the optimal number of colors. Nevertheless, several studies showed that in practice and in theory the Greedy algorithm performs poorly (Kučera, 1991, Turner, 1988). We only mention it here because our GA with order-based representation uses it as a decoder.

DSatur from (Brélaz (1979)) uses a heuristic to dynamically change the ordering of the nodes and then applies the Greedy method to color the nodes. It works as follows:

- A node with highest saturation degree (= number of differently colored neighbors) is chosen and given the smallest color that is still possible.
- In case of a tie, the node with highest degree (= number of neighbors that are still in the uncolored subgraph) is chosen.
- In case of a tie a random node is chosen.

Because of the random tie breaking, DSatur becomes a stochastic algorithm and just like for the GA, results of several runs need to be averaged to obtain useful statistics. For our investigation the backtrack version of Turner (1988) has been implemented, which backtracks to the lastly evaluated node that still has available colors to try. One search step is defined as expanding a node with a new color, including those done after a backtracking. The DSatur heuristics performs very well, therefore we use it as a heuristic technique to compare our EAs with.

Quite some research has been done about graph coloring with genetic algorithms, like Fleurent and Ferland who have successfully considered various hybrid algorithms in (Fleurent and Ferland (1996a)), and who have extended their study into a general implementation of heuristic search methods in (Fleurent and Ferland (1996b)). Others include von Laszewski who has looked at structured operators and has used an adaption step to improve the convergence rate of a genetic algorithm (Laszewski (1991)). Davis' algorithm in (Davis (1991)) was designed to maximize the total of weights of nodes in a graph colored with a fixed amount of colors. This resembles the problem definition used in the SAW-ing algorithm where we also add weights to the nodes of the graph that is being colored. The difference is that the SAW-ing algorithm uses variable weights to guide its search, while Davis' algorithm sees the fixed weights as the problem instance and then tackles this problem as an optimization problem. Coll, Durán, and Moscato (1995) discuss graph coloring and crossover operators in a more general context.

3. Performance measures of algorithms

The comparisons between different GAs and comparisons of GAs and DSatur will be based on two different measures: success rate and computational effort. Probabilistic algorithms may find a solution in one run and may not find one in another run. To cope with this problem we execute a number of runs and monitor how many times a solution is found. The measure *success rate* (SR) is the percentage of runs in which a solution is found, it gives an estimation of the probability of finding a solution in one run. The most obvious measure of computational effort for evaluating algorithms is the *time complexity*, e.g., CPU time in seconds (Kronsjo (1987)). A better option, however, is an implementation and hardware independent measure, *computational complexity*, i.e., the number of basic operations required to find a solution. For search algorithms this is the number of basic search steps a particular algorithm uses. Unfortunately, different search algorithms can use different search steps. For a GA a search step is the creation (and evaluation) of a new individual, in our case a new coloring. Thus, computational complexity will be measured by the average number of fitness evaluations in successful runs, denoted as AES (Average number of Evaluations to Solution). Fair as this measure may seem, even different GAs may not fully be comparable this way. For instance, a GA using a heuristic within the crossover operator performs ‘hidden labor’ with respect to a GA applying a standard ‘blind’ crossover. In other words, the extra computational efforts in performing a heuristic crossover are not visible if we compare the AES values. A search step of DSatur is a backtracking step, i.e., giving a node a new color. Thus, the computational complexity of DSatur is measured differently than that of a GA—a problem that is rooted in the different nature of the algorithms and cannot be circumvented. Despite of these drawbacks we compare algorithms by AES in the first part of Section 7, because this measure is independent from implementational and executional issues, such as the processor, programming language, network load, etc. In the second part of Section 7, however, we compare the scaling-up behavior of the investigated algorithms, i.e., we show how the AES values change with growing problem sizes. Regardless of the different meaning of AES for different algorithms this comparison is by all means fair.

4. Grouping Genetic Algorithm

The *Grouping Genetic Algorithm* (GGA) was introduced by Falkenauer and Delchambre (1992)) and further studied in (Falkenauer (1994, 1996)). The GGA tries to capture and exploit the structure of grouping problems by using an appropriate chromosomal representation and genetic operators. The outlines of the GGA are given in figure 1, while the different steps are explained in the remainder of this section.

In general, the grouping representation consists of two parts: an object part and a group part. The object part consists of n genes, where n is the number of objects to be grouped, the group part consists of a permutation of the k group labels. An object gene can take any of the k group labels as allele, indicating that the object in question belongs to group of the given label. In a graph coloring context objects are nodes and groups are colors; an example of a chromosome for $n = 6$ and $k = 3$ is shown in figure 2. The group part shows

```
Grouping GA
    Initialize population
    Evaluate population
    while not Stop-condition do
        Sort the population using 2-tournament selection
        Apply crossover to the best $N_c$ individuals
        Replace the worst $N_c$ with the offspring
        Mutate $N_m$ randomly selected individuals in the population
        Apply inversion to $N_i$ randomly selected individuals in the population
        Evaluate population
    end while
```

Figure 1. Pseudo code of the Grouping Genetic Algorithm.

$$\underbrace{\mathtt{BABBCA}}_{\text{objects}} : \underbrace{\mathtt{BAC}}_{\text{groups}}$$

Figure 2. Example of a chromosome in grouping representation.

that three are colors `A`, `B` and `C` are used for coloring the graph. The object part discloses that node 2 and 6 are colored with color `A`, nodes 1, 3 and 4 are colored with color `B` and node 5 is colored with color `C`.

The GGA uses genetic operators on the group part of the chromosomes, and adjusts the object part to the corresponding changes in the group part. We will use the three operators described in (Falkenauer (1994)), which are crossover, mutation and inversion. The crossover is by far the most difficult operator of these three, it uses a number of steps to compute two new chromosomes from two parents. These steps are the following:

1. Copy parent 1 to child 1 and copy parent 2 to child 2. Select at random two crossing sites, delimiting the crossing section, in each of the two parents' group part.

   Parent 1 : `BABBCA:B}A}C`
   Parent 2 : `acabba:}cb}a`

2. Inject the contents of the crossing section of the first parent before the first crossing site of the second child. Because we are working with the group parts, this means we are injecting groups of the first parent into the second child.

   Child 1 : `BABBCA:BAC`
   Child 2 : `acabba:Acba`

3. Overwrite the object part of child 2 such that membership of the newly inserted groups is enforced (inherited from parent 1).

   Child 1 : `BABBCA:BAC`
   Child 2 : `aAabbA:Acba`

4a. Adapt the resulting groups to satisfy the constraints of the problem. Here we throw away groups that have become empty or lost an object in step 3. This can result in objects which are not assigned to a group. These objects, which are marked by an `x`, are put in a queue.

   Child 1 : `BABBCA:BAC`
   Child 2 : `xAxbbA:Ab`
   Queue : $\{1, 3\}$

4b. We now have to reinsert the objects which reside in the queue. This can be done by any heuristic function, as long as the representation remains valid. We do this by looking at the node we want to insert and, if possible, assigning a color, by using a first-fit heuristic on a random order of the available colors, such that no constraints are violated. If this is not possible, we create a new group and assign the object to this group. Note, that this step can lead to an increase in the number of colors.

5. Execute the steps 3, 4 and 5 again, but with both parents exchanged.

The mutation operator uses a similar procedure. When a chromosome is selected for mutation, a number of elements in the group part are deleted. When a group is deleted all nodes in the object part having that color will be temporarily uncolored. After the deletion of groups we will reinsert the uncolored nodes using the same heuristic that is used in the crossover operator.

BABDCAEE : D|BEA|C $\rightarrow$ BABDCAEE : DAEBC

Figure 3. Example of inversion in grouping representation.

The third operator is inversion, which only operates on the group part of the chromosome, without affecting the object part. We randomly mark two points in the group part of the chromosome and then reverse the order of the groups between these two points. An example can be found in figure 3.

The crossover and mutation operators work in such a way that the length of the group part of a chromosome can vary between the minimal number of colors needed to color a graph and the number of nodes of the graph. We apply the algorithm to minimize the number of colors needed to color the graph. For the 3-colorable graph instances we investigate the minimum is known to be three. Therefore, we let the fitness of a chromosome depend on the number of 'unnecessary' colors and the amount of nodes which are colored with an 'unnecessary' color, where the three colors that color the most nodes are considered as 'necessary' and the remaining colors are defined as 'unnecessary'. For a formal definition of the fitness function let us assume that k is the minimal number of colors needed for coloring the graph. Furthermore, let a chromosome x use l colors $c_1, \ldots, c_l$ with $k \leq l$. Now let us define $g(x, y)$ as the function that returns the amount of nodes colored with color y in chromosome x. Without loss of generality we can assume that the colors c_i with $i \in \{1, \ldots, l\}$ are ordered in such a way that $g(x, c_1) \geq g(x, c_2) \geq \cdots \geq g(x, c_l)$. Now the fitness function can be defined as:

$$f_k(x) = l - k + \sum_{i=k+1}^{l} g(x, c_i) \tag{1}$$

To minimize this function the GGA must minimize the number of colors used for coloring the graph, it receives penalty points for each extra color and for every node colored with such a color. It is easy to see that it reaches its global minimum of zero, if and only if the amount of colors used is equal to the minimum amount of colors needed.

Initialization of the population is done by coloring every individual in the population using the first-fit heuristic and starting the coloring at a random node. The algorithm contains a pseudo-sorting procedure using 2-tournament selection. A pseudo-sorted list of individuals is created by repeatedly selecting two different individuals uniform randomly from the population to compete with each other. The loser, i.e., the one with the worse fitness value, is removed from the population and inserted at the top of the list. The procedure stops when the population becomes a singleton and the last remaining individual is added to the list. The individuals that were inserted first have 'sank' to the bottom of the list and are seen as the worst individuals, while those inserted last are the best ones. Note that the algorithm replaces the offspring of the best N_c individuals with the worst N_c individuals, therefore the population size must be at least $2N_c$. As for the parameter setting we compared the values recommended by Falkenauer (1994) (a population of fifty individuals, $N_c = 12$, $N_m = 4$ and $N_i = 4$, and using an allele mutation probability, giving the probability that an allele from the group part is being deleted when the individual is undergoing mutation, of 10%) with other values. The best option turned out to be using more mutation and less

Table 1. GA setup used in the experiments, except for the GGA.

GA type	Steady state
Selection	2-tournament
Deletion strategy	Worst-deletion
Crossover rate	1.0
Mutation rate	1/chrom.length
Stop-criterion	$T_{\max}$ fitness evaluations

crossover, in particular $N_c = 8$, $N_m = 12$. During the present investigation these values will be used.

5. Standard genetic algorithms

We experimented with several non-grouping genetic algorithms, varying different components. The common features of the GAs used are summarized in Table 1.

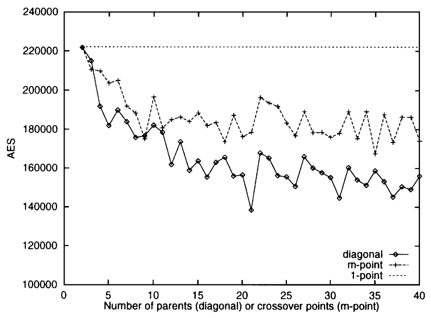
In the *integer representation* each gene of an individual represents a node in the graph and its value can be one of the three colors. Thus, the chromosome length L equals the number of nodes n in the given graph. The fitness function is based on penalizing constraint violation. We consider each edge as a constraint and in case of m edges the penalty function f is

$$f(x) = \sum_{i=1}^{m} w_i \cdot \chi(x, e_i) \tag{2}$$

where w_i is the penalty, or weight, assigned to the ith constraint (edge e_i) and

$$\chi(x, e_i) = \begin{cases} 1 & \text{if } x \text{ violates } e_i, \text{ i.e., } e_i = (k, l) \text{ and } x_k = x_l \\ 0 & \text{otherwise} \end{cases}$$

It is obvious that when the minimum penalty of zero is reached, no constraints are violated and a solution is found. Here we use the same penalty $w_i \equiv 1$ for each constraint (edge), thus f simply counts the violated constraints. It has been conjectured by Falkenauer that this representation and the corresponding genetic operators are not well-suited for grouping problems, such as graph coloring, mainly because of the 'blind disruption' of chromosomes (Falkenauer (1994)). Our experiments confirmed this conjecture, but one unexpected result deserves special attention. Namely, increasing the disruptiveness of crossover operators increased GA performance, seemingly contradicting that crossover is harmfull. Detailed presentation of all results would consume too much space, a full overview can be found in (Eiben and Van der Hauw (1996)). Here we only give an illustration in figure 4 that illustrates that increasing the number of crossover points in multi-point crossover (De Jong and Spears (1992)), and increasing the number of parents in multi-parent crossovers (Eiben, Raué, and Ruttkay (1994)), leads to better results.



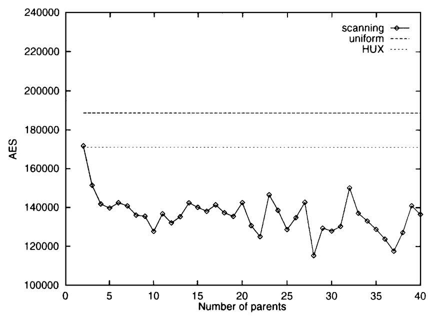
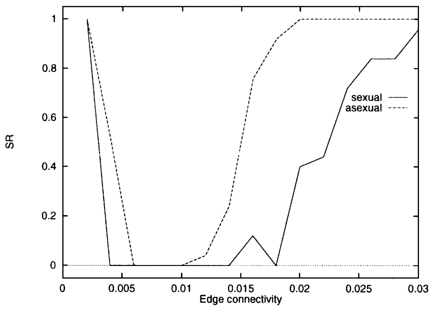


Figure 4. Integer representation: effect of more crossover points and more parents on AES for $G_{eq,n=1000,p=0.025,s=5}$. $T_{\max} = 250000$ in each run, the results are averaged over 25 independent runs for each setting (number of crossover points, number of parents).



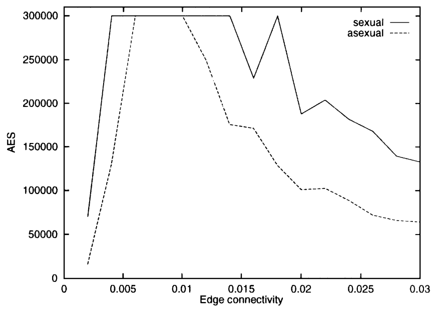


Figure 5. Integer representation: asexual (only mutation) vs. sexual reproduction (uniform crossover + mutation + incest prevention) for $n = 1000$ near the phase transition. $T_{\max} = 300000$ in each run, the results are averaged over 25 independent runs on each instance.

Nevertheless, the best GA variant within this representation turned out to be an asexual GA using only mutation and no crossover in a $(1 + 1)$ scheme, thus with population size 1 and preservative selection. Figure 5 shows a comparison of the best asexual (mutation only) and sexual (using crossover and mutation) variants.

In *order-based* GAs the individuals are permutations and special operators are used to recombine and mutate permutations, (Starkweather et al. (1991), Fox and McMahon (1991)). For graph coloring we define chromosomes as permutations of nodes and apply a decoder that constructs a coloring from a permutation. So just as for integer representation the chromosome length is $L = n$. As a decoder we have used the Greedy Algorithm which colors a node with the lowest color that does not violate constraints and leaves nodes uncolored when no colors are left. The simplest way of evaluating a permutation is then to use the number of uncolored nodes in the coloring belonging to it. Formally, we use a penalty function that concentrates on the nodes, instead of the edges. The function f is now defined as:

$$f(x) = \sum_{i=1}^{n} w_i \cdot \chi(x, i) \tag{3}$$

where w_i is now the penalty (or weight) assigned to node i and

$$\chi(x, i) = \begin{cases} 1 & \text{if node } x_i \text{ is left uncolored because of a constraint violation} \\ 0 & \text{otherwise} \end{cases}$$

Just like before, we use $w_i \equiv 1$, but while for integer representation the fitness function counted the 'wrong' edges, here we count the uncolored nodes. Note that the search space for order-based representation is much bigger than for the integer representation: $n!$ instead of 3^n. Because at most 3^n different possible colorings exist this coding is highly redundant. Additionally, the penalty function given in Eq. (3) supplies less information than the one given in Eq. (2). These considerations might suggest that this GA will be less successful, but the experiments show that the opposite is true.

Using this representation we compared different operators, various population sizes and the effect of using only mutation. The outcomes are similar to those of integer representation, in the sense that an asexual GA using only SWAP mutation and no crossover in a $(1 + 1)$ scheme outperforms the best sexual GA using OX2 crossover together with SWAP on large graphs, see figure 6 for illustration. On small graphs this is only partly true, but the global conclusion is that the asexual GA has the best overall performance and that order based representation is superior to integer representation.

Summarizing our findings on using GAs with traditional representations we can note the following. The best performance is achieved with an algorithm without crossover, exclusively using mutation and having population size 1. The question arises whether such an algorithm still can be seen as genetic. A 'no' is supported by noting that the presence of crossover and a population with more than one elements is crucial for GAs. However, one could also argue that the representation and the mutation is standard, only the parameter setting is extreme. To avoid conflicts with conventions on terminology we percieve and name the winning algorithm variant as an evolutionary algorithm. To this end note, that in contemporary evolutionary computation the term evolutionary algorithm comprises among others genetic algorithms, evolution strategies (ES), and evolutionary programming (EP) (Bäck, Fogel, and Michalewicz (1997)), and that $(1 + 1)$ style alorithms are

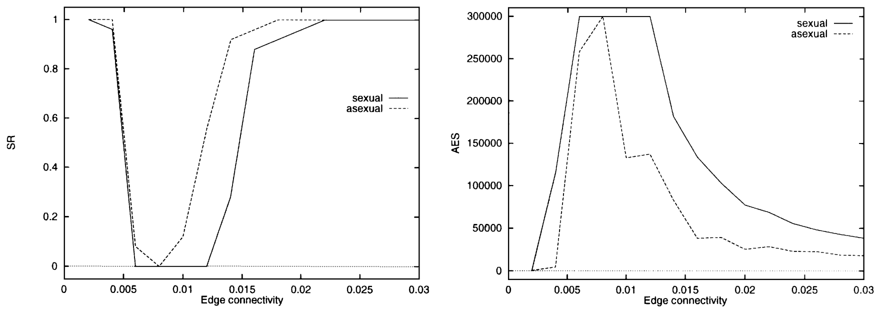


Figure 6. Order-based representation: asexual (only SWAP mutation) vs. sexual reproduction (OX2 + SWAP mutation) for $n = 1000$ near the phase transition. $T_{\max} = 300000$ in each run, the results are averaged over 25 independent runs on each instance.

common in ES (Schwefel (1995)) and EP always operates without using crossover (Fogel (1995)).

6. Stepwise adaptation of weights

Evolutionary algorithms are intrinsically dynamic and adaptive processes. It is thus rather unnatural to use static control parameters that remain constant during the evolution. Using constant parameters is, however, the traditional practice. In the meanwhile, there is an increasing number of papers that consider varying parameter settings. This accumulates knowledge on a research area that is thus becoming an emerging sub-technology within evolutionary computation. Looking at related work on varying parameters in EAs one can distinguish common features several approaches share. The classification in the paper by (Hinterding, Michalewicz, and Eiben (1997)) distinguishes three different streams. *Dynamic parameters* obtain different values along the evolution prescribed by a user defined schedule. This schedule typically assigns new values to parameters depending on time, most commonly expressed by the number of generations or fitness evaluations. *Adaptive parameters* obtain new values by a feedback mechanism that monitors the evolution. The new values typically depend on the achieved progress. This progress measure is the input of a mechanism that resets the parameter. *Self-adaptive parameters* are encoded in the chromosomes and undergo evolution themselves. As opposed to the previous two techniques, here there is not even an indirect user control of this parameter. It is clear that varying parameters suits the general ‘evolutionary spirit’ better than static ones, furthermore, they have technical advantages. First of all, they often lead to increased performance. Second, adaptive and self-adaptive parameters free the user from determining these parameters by having the EA doing it. This reduces the chance for incorrect parameter settings.

Our approach falls in the adaptive category in the above classification scheme. Technically it amounts to modifying the weights of components of the penalty function, hence modifying the fitness landscape during the search, based on feedback from the actual population. This technique was introduced for constraint solving with GAs in (Eiben, Raué, and Ruttkay (1995b)), where the constraints were weighted. The rationale behind it is clear: satisfying a constraint with a high penalty gives a relatively high reward to the algorithm, hence it will be ‘more interested’ in satisfying such constraints. Thus, using appropriate weights focuses the ‘attention’ of the algorithm, hence it can improve the performance. Appropriate in this case means that the constraint weights should reflect how important, or rather, how difficult a specific constraint is. This causes two problems. On the one hand, to determine relative hardness of constraints can require substantial knowledge on the problem. On the other hand, ‘being hard’ cannot be seen independently from the applied problem solver. Therefore, it is a natural idea to leave the decision on measures of hardness to the problem solver itself. Although there are well-founded arguments (Culberson (1996), Wolpert and Macready (1997)) stating that no search technique can be superior *in general*, having an evolutionary algorithm adjusting its parameters itself has been proved to be powerful under many circumstances (Angeline (1995), Hinterding, Michalewicz, and Eiben (1997)). In the *particular* case of constraint satisfaction problems we expect that a mechanism enabling the GA to learn weights itself can circumvent difficulties of setting these weights by a human user.

It is interesting to note that using adaptive, or learning features to improve search performance is done for other types of algorithms too. The breakout method of Morris (1993) is the earliest example we know of. Adaptive memory features as advocated by Glover (1996), and applied by Løkketangen and Glover (1996), are close to the 'evolutionary spirit' and work very well on constrained problems. Also, variants of the the GSAT algorithm for satisfiability problems by Selman and Kautz (1993) and Frank (1996a, 1996b), apply mechanisms that re-evaluate weights of clauses during the search process.

The best EA for our graph 3-coloring problem we found so far uses order-based representation penalizing uncolored nodes. We extend this variant with an adaptive mechanism, thus the basic idea is now implemented by monitoring which variables in the best individual violate constraints and raising the penalty w_i belonging to these variables. Depending on when the weights are updated we can distinguish an off-line (after the run) and an on-line (during the run) version of this technique, see figures 7 and 8.

In (Eiben, Raué, and Ruttkay (1995a), Eiben and Ruttkay (1996)) the off-line version was applied, here we will use the on-line version modifying the weights, hence modifying the fitness function, during the evolution. After a certain period, in particular T_p fitness evaluations, the best individual in the population is colored and the weights of its uncolored nodes (that are considered hard for the EA) are increased by Δw, i.e., w_i is set to $w_i + \Delta w$. This implies that the EA has to search on a dynamically changing fitness landscape, consequently the population has to be re-evaluated after each period.[3] We call this mechanism Stepwise Adaptation of Weights (SAW).

It is very interesting to see the fitness curve of a run of the EA with the SAW mechanism. Figure 16 shows a run when a solution is found. The left curve has a higher resolution, displaying the fitness of the best individual between 0–10000 evaluations, the right curve shows the range 0–80000. The higher resolution curve (left) shows that within each period

```
Off-line saw
    set initial weights (thus fitness function f)
    for x test runs do
        run the GA with this f
        redefine f after termination
    end for
```

Figure 7. Pseudo code of the off-line weight adaptation mechanism.

```
On-line saw
    set initial weights (thus fitness function f)
    while not termination do
        for the next Tp fitness evaluations do
            let the GA go with this f
        end for
        redefine f and recalculate fitness of individuals
        end while
```

Figure 8. Pseudo code of the on-line weight adaptation mechanism.

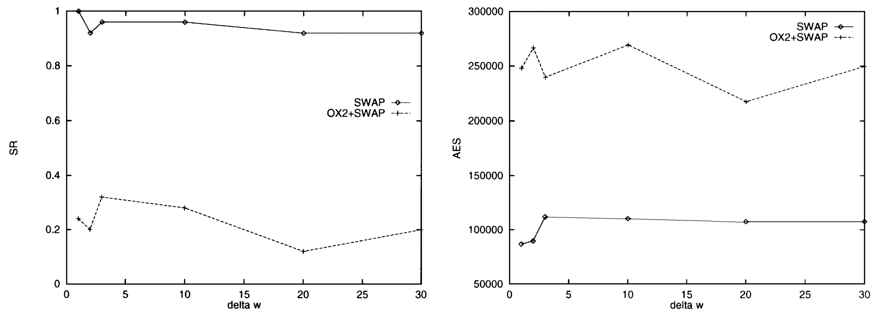


Figure 9. Effect of varying Δw on $G_{eq,n=1000,p=0.010,s=5}$. $T_p = 250$, $T_{\max} = 300000$. SWAP uses population size 1, OX2 + SWAP uses population size 700.

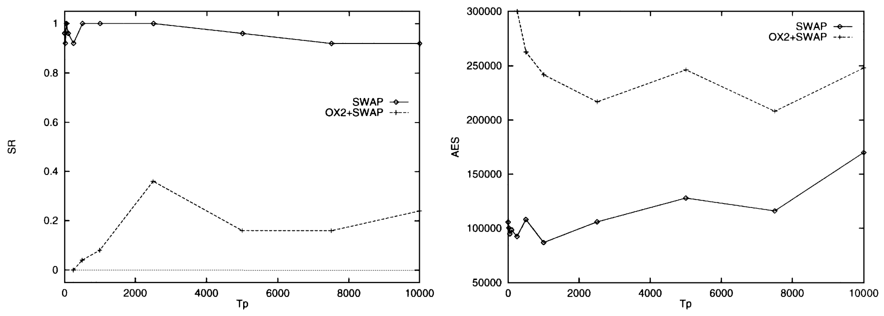


Figure 10. Effect of varying T_p on $G_{eq,n=1000,p=0.010,s=5}$. $\Delta w = 1$, $T_{\max} = 300000$. SWAP uses population size 1, OX2 + SWAP uses population size 700.

the penalty drops as the EA is making progress and then sharply rises when the weights are updated, giving the image of a **SAW!**

Note that the SAW mechanism introduces two new parameters, T_p and Δw. It is thus important to check whether the performance of a SAW-ing EA is sensitive for the parameter values. To this end we performed an extensive test series that showed that the performance is pretty much independent from these parameters. As an illustration we present the outcomes for $n = 1000$, for the asexual as well as for the sexual order-based EA in figures 9 and 10. The exact values for Δw and T_p do not have a significant effect on the performance, as long as T_p is significantly smaller than $T_{\max}$. From now on, we will use $\Delta w = 1$ and $T_p = 250$ for no specific reason. Figures 9 and 10 show that, also in the presence of the SAW mechanism the sexual EA using OX + SWAP is clearly inferior to the asexual variant.

The effect of the SAW mechanism on the EA performance on graphs with $n = 1000$ and $p = 0.010$ can be seen in Table 2. The increase in performance after adding the SAW-ing mechanism is dramatic: the success rate averaged over all seeds raises from 9% to 92%, while the number of search steps drops from 205643 to 89277. The figures show not only that a SAW-ing EA highly outperforms the other techniques, but also that the performance is rather independent from the random seeds. Thus, the SAW mechanism is not only highly

Table 2. Comparing DSatur with backtracking, the Grouping GA, $(1+1)$ order-based GA using SWAP and $(1+1)$ order-based GA using SWAP and the SAW mechanism ($T_p = 250$, $\Delta w = 1$) for $n = 1000$, $p = 0.010$, different seeds.

	$s = 0$		$s = 1$		$s = 2$		$s = 3$		All 4 seeds	
	SR	AES	SR	AES	SR	AES	SR	AES	SR	AES
DSatur	0.08	125081	0.00	300000	0.00	300000	0.80	155052	0.22	220033
GGA	0.00	300000	0.00	300000	0.00	300000	0.00	300000	0.00	300000
GA	0.24	239242	0.00	300000	0.00	300000	0.12	205643	0.09	261221
GA + SAW	0.96	76479	0.88	118580	0.92	168060	0.92	89277	0.92	113099

effective, obtaining much better success rates at lower costs, but also very robust. Somewhat surprising is the low performance of the GGA. On average it terminates with 5 colors and approximately 50 nodes in the 'unnecessary' colors.

7. Comparing the GGA, the SAW-ing EA and DSatur

The results summarized in Table 2 serve as an indication of the viability of our SAW-ing mechanism. For a solid conclusion on the performance of the SAW-ing EA we perform an extensive comparison between the Grouping GA as described in Section 4, DSatur with backtracking, and a SAW-ing EA with order-based representation, simple greedy decoder, OneSWAP mutation[4] and no crossover in a $(1+1)$ selection scheme using $T_p = 250$, and $\Delta w = 1$ for the SAW mechanism. The comparison is performed on arbitrary 3-colorable, equi-partite 3-colorable and flat 3-colorable graphs for $n = 200$, $n = 500$ and $n = 1000$ with edge connectivities around the phase transition. The results are based on 100, 50 and 25 runs for $n = 200$, $n = 500$ and $n = 1000$, respectively. For all instances, the graph is generated with seed $s = 5$. DSatur and both GAs are allowed $T_{\max} = 300000$ search steps.

The test results concerning SR and AES values are given in the figures 11–13. The Grouping GA is inferior to the other two algorithms on all graph instances. On small graphs ($n = 200$) DSatur is better for all p's than the SAW-ing EA, except for flat graphs near the phase transition, see figure 13 for $n = 200$. On medium size graphs ($n = 500$) the SAW-ing EA is slightly better on arbitrary and equi-partite topologies. On flat graphs we see that the performance of the SAW-ing EA deteriorates much less at the phase transition than that of DSatur. On large graphs ($n = 1000$) the SAW-ing EA is clearly better w.r.t. success rates, because it is often able to find solutions where DSatur does not find any. The AES curves are sometimes crossing, but in general the SAW-ing EA needs fewer steps. The reason for these differences could be that the instances with $n = 200$ are small enough for DSatur to get out of local optima and find solutions, while this is not possible anymore for $n = 500$ and $n = 1000$ where the search space becomes too big. Evaluating the overall usefulness of the SAW-ing EA for graph coloring one has to consider two cases. On the easy problems, i.e., small graphs and far from the phase transition DSatur is better. On the hard instances near the phase transition and on large graphs the SAW-ing EA outperforms

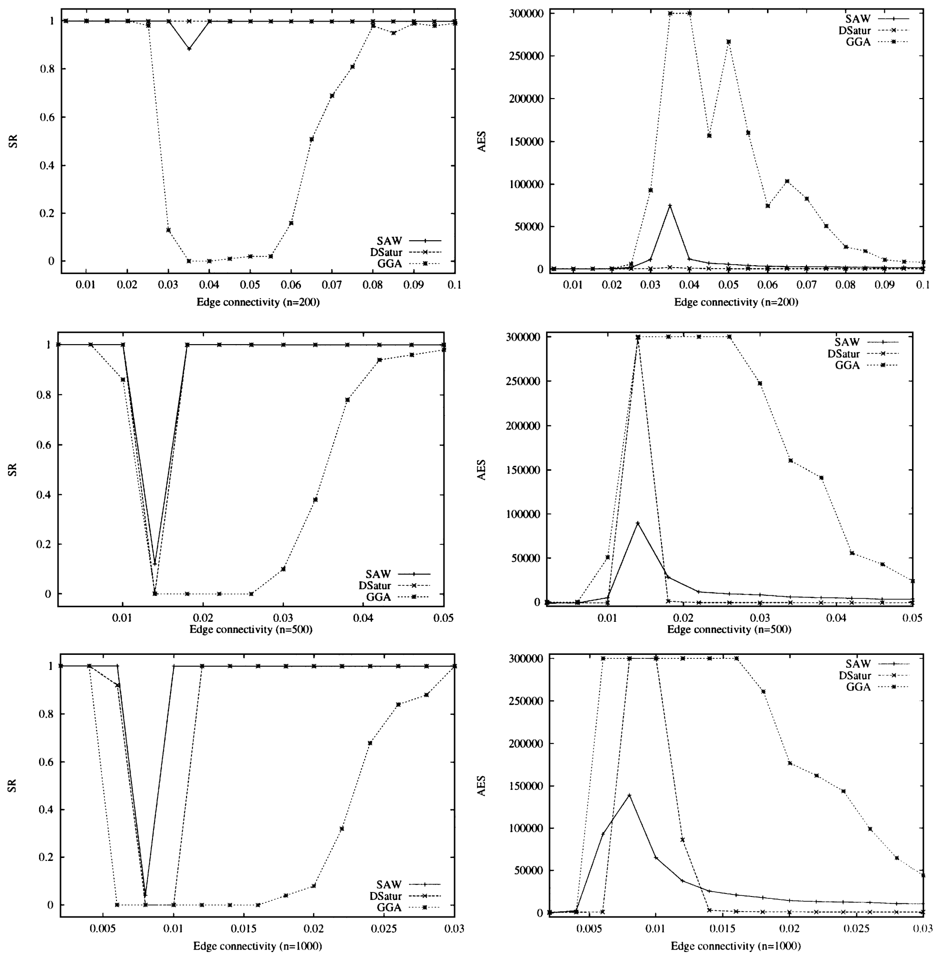


Figure 11. Comparison for arbitrary 3-colorable graphs, for $n = 200$, $n = 500$ and $n = 1000$.

DSatur. These results are very good in the light of the fact that DSatur is a highly problem tailored graph coloring algorithm, whereas the SAW-ing EA is a general purpose algorithm for constraint satisfaction, using no specific domain knowledge on graph coloring. Another strong property of the SAW-ing EA is that it can take more advantage of extra time given for search. We increased the total number of evaluations to 1000000 and observed that DSatur still had $\mathrm{SR} = 0.00$ on $G_{eq,n=1000,p=0.008,s=5}$. The performance of the SAW-ing EA, however, increased from $\mathrm{SR} = 0.00$ to $\mathrm{SR} = 0.44$ ($\mathrm{AES} = 407283$), showing that it is able to benefit from more time given, where the extra time for backtracking is not enough to get out of the local optima.

It is an important question how the performance of an algorithm changes if the problem size grows. This is the issue of scalability, which we will consider w.r.t. to the success rates and the computational complexity. Thus, while the figures 11–13 were drawn for fixed n's

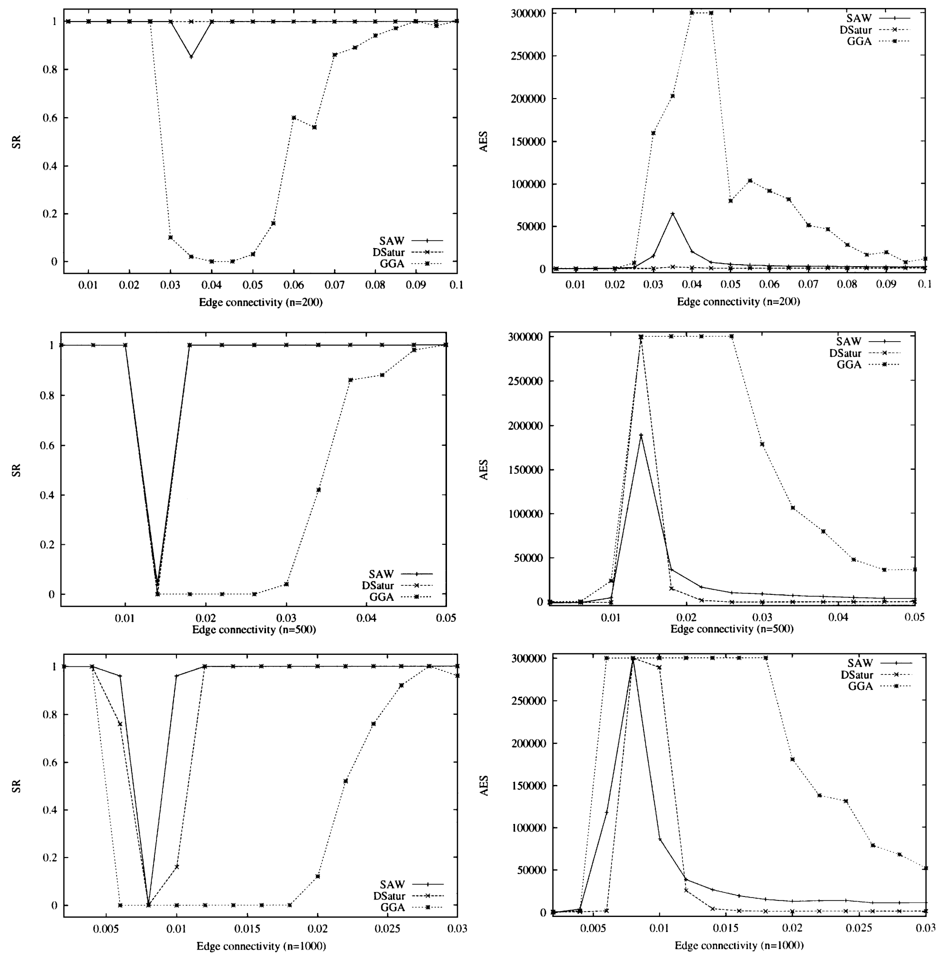


Figure 12. Comparison for equi-partite 3-colorable graphs, for $n = 200$, $n = 500$ and $n = 1000$.

and varying p's, in the sequel we vary n (and keep $p = c/n$ for some constant c related to it). We tested the three algorithms using the same parameters as before for $p = 8.0/n$ at the phase transition. The results are given in figure 14 showing that the Grouping GA could only solve the smallest problem instances and DSatur was not able to find solutions on larger problem instances. Therefore, we also made a comparison on easier instances for $p = 10.0/n$. Figure 15 exhibits these results showing that the SAW-ing EA scales up much better than the other two methods.

As discussed in Section 3 even the implementation independent number of search steps is not ideal for comparing different algorithms. Nevertheless, comparing the growth rate of the number of search steps when the problem size (the number of nodes in the graphs to be colored) grows gives a sound basis for judgment, even though the absolute meaning of the given numbers may differ. The scale-up curves for AES on the right-hand side plots

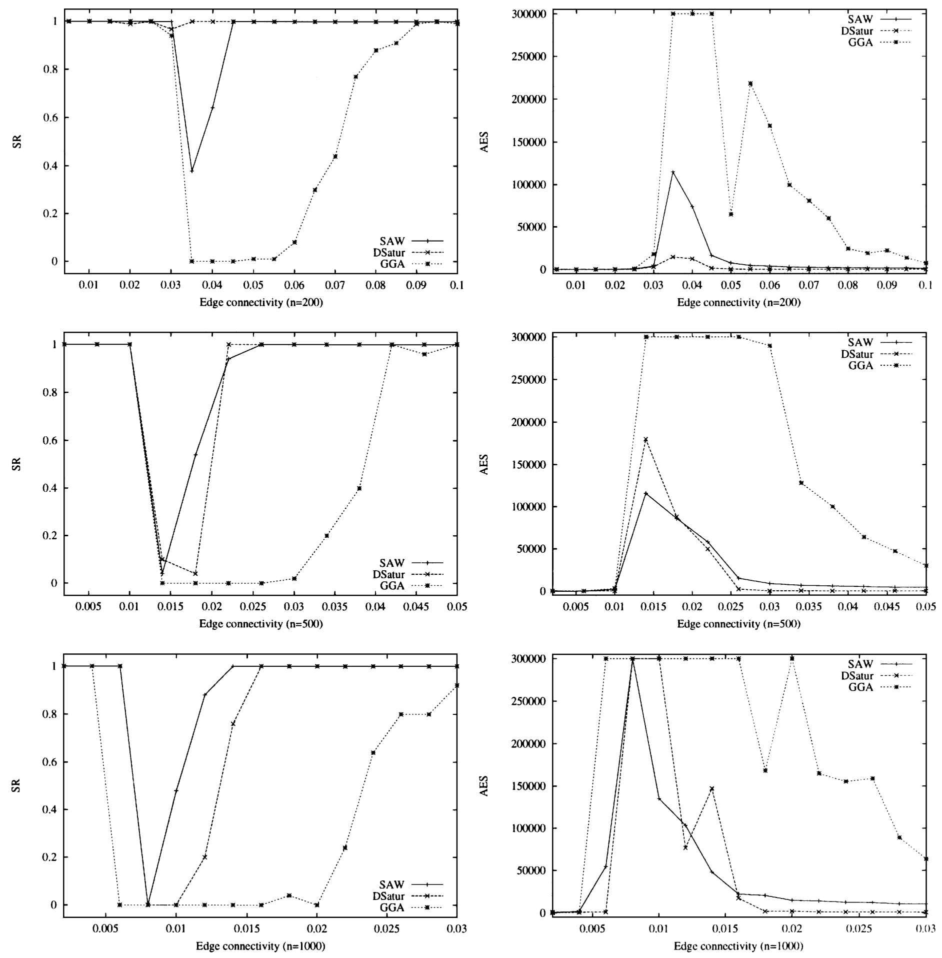


Figure 13. Comparison for flat 3-colorable graphs, for $n = 200$, $n = 500$ and $n = 1000$.

of figure 14 ($p = 8.0/n$) and figure 15 ($p = 10.0/n$) show interesting results. On the easier case, for $p = 10.0/n$, DSatur is faster up to $n = 750$, but the SAW-ing EA scales up much better. On the really hard graphs, for $p = 8.0/n$, the SAW-ing EA outperforms DSatur already from $n = 250$. The curves for both edge connectivity values indicate that the SAW-ing EA scales up linearly with the problem size.

8. Conclusions

NP-complete problems, such as graph coloring, form a big challenge for designers of algorithms in general. For evolutionary algorithms, in particular, constraint satisfaction problems form a specific challenge. In this paper we investigated how EAs can be applied for graph 3-coloring.

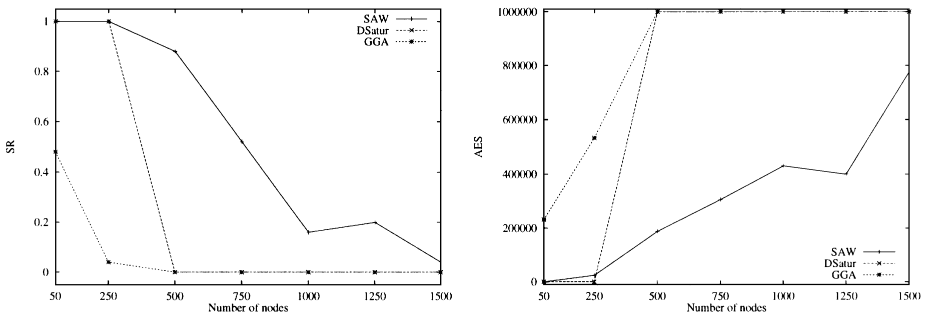


Figure 14. Scale-up curves for SR and AES on equi-partite graphs with $p = 8.0/n$ based on $T_{\max} = 1000000$ and 25 runs.

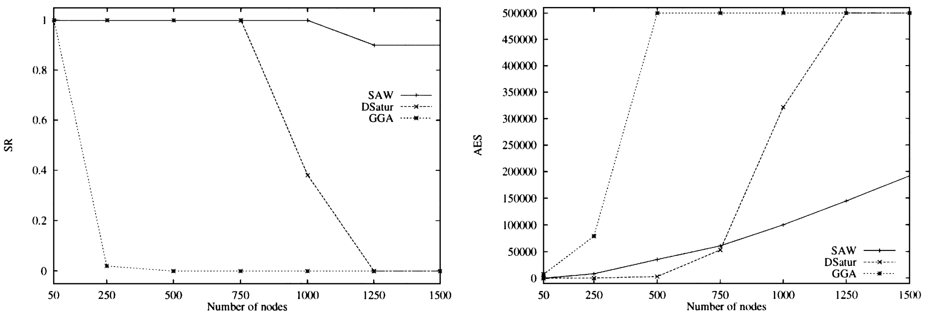


Figure 15. Scale-up curves for SR and AES on equi-partite graphs with $p = 10.0/n$ based on $T_{\max} = 500000$ and 50 runs.

After trying several traditional genetic algorithm variants based on integer and order-based representation we concluded that using only mutation is better than using mutation and crossover. This fact is in accordance with the common opinion in evolutionary computation that traditional genetic crossovers are not appropriate for so-called grouping problems, because they ‘blindly’ disrupt chromosomes. However, it is interesting to note that in integer representation using more crossover points and more parents—which both imply heavier mixing than usual operators—increases the performance, cf. Section 5.

As discussed at the end of Section 5, it can be argued that switching off crossover and setting the population size to 1 in a genetic algorithm results in an algorithm that cannot be called genetic anymore. Iterated stochastic hill-climbing could be an appropriate name, but we rather use the name evolutionary algorithm. This name reflects the origin of the method and there are other algorithms in evolutionary computation that use exclusively mutation, or population size 1 (Fogel (1995), Schwefel (1995)).

Following the recommendations that on grouping problems special representations and crossovers are advisable (Falkenauer (1994)), we have implemented and tested the grouping GA. Surprisingly, the GGA is inferior to a simple order-based asexual EA, see Table 2. This,

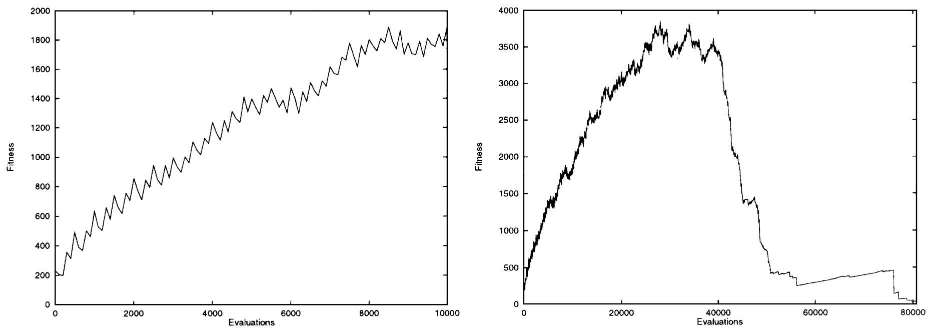


Figure 16. Fitness curve development during a run of the SAW-ing EA ($n = 1000$, $p = 0.010$). Left: first 10000 evaluations, right: whole run of 80000 evaluations.

suggests that using specific grouping operators is not the only solution for handling the graph coloring problem. Using only mutation and population size 1 works as well.

The main research subject of this investigation is the technique called Stepwise Adaptation of Weights. This mechanism is problem independent and it amounts to repeatedly redefining the weights that are used in the definition of the fitness (penalty) function. Similar mechanisms have been applied in other search algorithms; here we use it in an EA. Adding the SAW mechanism to the simple order-based asexual EA increased its performance with a factor 10 (success rates increased from 9% to 92%), and made it the best algorithm on the problem instances we used for a preliminary comparison, cf. Table 2.

Conducting an extensive series of experiments on three different types of graph, three different sizes and various edge connectivity values confirmed that the SAW-ing EA is a powerful graph coloring algorithm. The SAW-ing EA was better than the GGA and also outperformed DSatur by three criteria: (1) on the hardest graph instances it performs better than DSatur, (2) it is able to increase its performance when given more time, whereas DSatur is not, (3) it scales up much better, indicating a linear scale-up w.r.t. computational complexity. Especially nice about the SAW-ing EA is that it is *not* tailored to the problem of graph coloring. Our findings are thus relevant in the broader context of evolutionary constraint handling. The SAW mechanism helps to circumvent a major problem of penalty based evolutionary constraint handling techniques by letting the GA find the right constraint weights.

Further research concerns application of SAW-ing to other (NP-complete) constrained problems and investigating variations of the SAW mechanism (Eiben and van der Hauw (1997)). Straightforward modifications are varying the value of Δw over the constraints, and allowing decreasing w_i for constraints that are already satisfied. Especially interesting to study is the learning procedure based on combining frequency and recency based memory of Løkketangen and Glover (1996), where the changes in weights are related to solution quality.

Finally, let us make an additional note on the constraint weights the EA finds. The plots in figure 16 suggest that problem solving with the SAW mechanism happens in two phases. In the first phase the EA is learning a good setting for the weights. In this phase the penalty increases a lot because of the increased weights. In the second phase the EA is solving the problem, exploiting the knowledge (appropriate weights) learned in the first phase. In this phase the penalty drops sharply indicating that using the right weights (appropriate fitness function) in the second phase the problem becomes ‘easy’. This interpretation of the fitness curves is plausible. We, however, do not want to suggest that the EA could learn universally good weights for a given graph instance. In the first place, another problem solver might need other weights to solve the same problem. Besides, in a control experiment we applied the SAW-ing EA to a graph, thus learning good weights, and then applied the EA to the same graph again using the learned weights non-adaptively, i.e., keeping them constant along the evolution. The results showed *worse* performance than in the first run when adaptive weights were used. Suggesting that the SAW mechanism works by enabling the problem solver to discover some hidden, universally good weights is, therefore, wrong. This seems to contradict the interpretation that distinguishes two phases of search. At the moment we tend to see the main advantage of SAW in allowing the EA to shift the focus of search (quasi) continuously and thus allowing implicit problem decomposition that guides the population through the search space.

Notes

1. Source code in C is available via `ftp://ftp.cs.ualberta.ca/pub/GraphGenerator/generate.tar.gz`.
2. The colors are ordered just to make selection possible.
3. Hereby the total number of fitness evaluations will not equal the total number of generated individuals. It could be argued that the number of re-evaluations should be included in the total number of evaluations. However, we want to count the search steps by the total number of generated colorings. Besides, re-evaluation is computationally cheap: the individual need not be decoded again, only the sum in Eq. (3) has to be re-computed for its uncolored nodes. Therefore, the number of re-evaluations is not included in the total cost.
4. Additional tests (not reported here) showed that OneSWAP which always swaps exactly one pair of genes is slightly better for the $(1 + 1)$ GA than usual SWAP.

References

Angeline, P. (1995). “Adaptive and Self-Adaptive Evolutionary Computation.” In M. Palaniswami, Y. Attikiouzel, R.J. Marks, D. Fogel, and T. Fukuda (eds.), *Computational Intelligence: A Dynamic System Perspective*. IEEE Press, pp. 152–161.

Arora, S., C. Lund, R. Motwani, M. Sudan, and M. Szegedy. (1992). “Proof Verification and Hardness of Approximation Problems,” *Proceedings 33rd IEEE Symposium on the Foundations of Computer Science*. IEEE Computer Sociecty, Los Angeles, CA, pp. 14–23.

Bäck, T., D. Fogel, and Z. Michalewicz. (eds.) (1997). *Handbook of Evolutionary Computation*. Bristol: Institute of Physics Publishing and New York: Oxford University Press.

Belew, R. and L. Booker. (eds.) (1991). *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann.

Blum, A. (1989). “An $O(n^{0.4})$-Approximation Algorithm for 3-Coloring (and Improved Approximation Algorithms for k-Coloring),” *Proceedings of the 21st ACM Symposium on Theory of Computing*. New York, ACM, pp. 535–542.

Brélaz, D. (1979). “New Methods to Color Vertices of a Graph,” *Communications of the ACM* 22, 251–256.

Cheeseman, P., B. Kenefsky, and W.M. Taylor. (1991). “Where the Really Hard Problems Are.” In J. Mylopoulos, and R. Reiter (eds.), *Proceedings of the 12th IJCAI-91*. Morgan Kaufmann, vol. 1, pp. 331–337.

Clearwater, S. and T. Hogg. (1996). “Problem Structure Heuristics and Scaling Behavior for Genetic Algorithms,” *Artificial Intelligence* 81, 327–347.

Coll, P., G. Durán, and P. Moscato. (1995). “A Discussion on Some Design Principles for Efficient Crossover Operators for Graph Coloring Problems,” *Anales del XXVII Simposio Brasileiro de Pesquisa Operacional*.

Culberson, J. (1996). “On the Futility of Blind Search,” Technical Report TR 96-18, The University of Alberta.

Culberson, J. and F. Luo. (1996). “Exploring the *k*-Colorable Landscape with Iterated Greedy.” In D. Johnson and M. Trick (eds.), *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. American Mathematical Society, pp. 245–284. Available by http://web.cs.ualberta.ca/˜joe/.

Davis, L. (1991). “Order-Based Genetic Algorihms and the Graph Coloring Problem.” *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold, pp. 72–90.

De Jong, K. and W. Spears. (1992). “A Formal Analysis of the Role of Multi-point Crossover in Genetic Algorithms,” *Annals of Mathematics and Artificial Intelligence* 5, 1–26.

Eiben, A., P.-E. Raué, and Z. Ruttkay. (1994). “Genetic Algorithms with Multi-parent Recombination.” In Y. Davidor, H.-P. Schwefel, and R. Männer (eds.), *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, number 866 in *Lecture Notes in Computer Science*. Springer-Verlag, pp. 78–87.

Eiben, A., P.-E. Raué, and Z. Ruttkay. (1995a). “Constrained Problems.” In L. Chambers (ed.), *Practical Handbook of Genetic Algorithms*. CRC Press, pp. 307–365.

Eiben, A., P.-E. Raué, and Z. Ruttkay. (1995b). “GA-Easy and GA-Hard Constraint Satisfaction Problems.” In M. Meyer (ed.), *Proceedings of the ECAI-94 Workshop on Constraint Processing*, number 923 in *Lecture Notes in Computer Science*. Springer-Verlag, pp. 267–284.

Eiben, A. and Z. Ruttkay. (1996). “Self-adaptivity for Constraint Satisfaction: Learning Penalty Functions,” *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*. IEEE Press, pp. 258–261.

Eiben, A. and J. van der Hauw. (1996). “Graph Coloring with Adaptive Evolutionary Algorithms,” Technical Report TR-96-11, Leiden University. Also available as http:// www.wi.leidenuniv.nl/˜gusz/graphcol.ps.gz.

Eiben, A. and J. van der Hauw. (1997). “Solving 3-SAT by GAS Adapting Constraint Weights,” *Proceedings of the 4th IEEE Conference on Evolutionary Computation*. IEEE Press, pp. 81–86.

Eiben, A. and Z. Ruttkay. (1997). “Constraint Satisfaction Problems.” In T. Bäck et al. (eds.), *Handbook of Evolutionary Computation*. Bristol: Institute of Physics Publishing, and New York: Oxford University Press.

Falkenauer, E. (1994). “A New Representation and Operators for Genetic Algorithms Applied to Grouping Problems,” *Evolutionary Computation* 2(2), 123–144.

Falkenauer, E. (1996). “A Hybrid Grouping Genetic Algorithm for Bin Packing,” *Journal of Heuristics* 2, 5–30.

Falkenauer, E. and A. Delchambre. (1992). “A Genetic Algorithm for Bin Packing and Line Balancing,” *Proceedings of the IEEE 1992 Int. Conference on Robotics and Automation*. IEEE Computer Society Press, pp. 1186–1192.

Fleurent, C. and J. Ferland. (1996a). “Genetic and Hybrid Algorithms for Graph Coloring.” In I.H.O. G. Laporte and P.L. Hammer (eds.), *Annals of Operations Research*, number 63 in *Metaheuristics in Combinatorial Optimization*. J.C. Baltzer AG, Science Publishers, pp. 437–461.

Fleurent, C. and J. Ferland. (1996b). “Object-Oriented Implementation of Heuristic Search Methods for Graph Coloring, Maximum Clique, and Satisfiability.” In M.A. Trick and D.S. Johnson (eds.), *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, pp. 619–652.

Fogel, D. (1995). *Evolutionary Computation*. IEEE Press.

Fox, B. and M. McMahon. (1991). “Genetic Operators for Sequencing Problems.” In G. Rawlins (ed.), *Foundations of Genetic Algorithms*. Morgan Kaufmann, pp. 284–300.

Frank, J. (1996a). “Learning Short-term Weights For GSAT,” Technical Report, University of California at Davis. Available by http://rainier.cs.ucdavis.edu/˜frank/decay.ml96.ps.

Frank, J. (1996b). “Weighting for Godot: Learning Heuristics For GSAT,” *Proceedings of the 13th AAAI-96*. AAAI / The MIT Press, pp. 338–343.

Garey, M. and D. Johnson. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freedman and Co.

Glover, F. (1996). “Tabu Search and Adaptive Memory Programming—Advances, Applications, and Challenges,” *Interfaces in Computer Science and Operations Research*. Norwell, MA: Kluwer Academic Publishers, pp. 1–75.

Grimmet, G. and C. McDiarmid. (1975). "On Colouring Random Graphs," *Mathematical Proceedings of the Cambridge Philosophical Society* 77, 313–324.

Hinterding, R., Z. Michalewicz, and A. Eiben. (1997). "Adaptation in Evolutionary Computation: A Survey," *Proceedings of the 4th IEEE Conference on Evolutionary Computation*. IEEE Press, pp. 65–69.

Johnson, D., C. Aragon, L. McGeoch, and C. Schevon. (1991). "Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning," *Operations Research* 39(3), 378–406.

Kronsjo, L. (1987). *Algorithms: Their Complexity and Efficiency*. Wiley and Sons, second edition.

Kučera, L. (1991). "The Greedy Coloring is a Bad Probabilistic Algorithm," *Journal of Algorithms* 12, 674–684.

Laszewski, G.V. (1991). "Intelligent Structural Operators For the *k*-Way Graph Partitioning Problem." In R. Belew and L. Booker (eds.), *Proceeding of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann, pp. 45–52.

Løkketangen, A. and F. Glover. (1996). "Surrogate Constraint Methods with Simple Learning for Satisfiability Problems." In D. Du, J. Gu, and P. Pardalos (eds.), *Proceedings of the DIMACS workshop on Satisfiability Problems: Theory and Applications*. American Mathematical Society.

Morris, P. (1993). "The Breakout Method for Escaping From Local Minima," *Proceedings of the 11th National Conference on Artificial Intelligence, AAAI-93*. AAAI Press/The MIT Press, pp. 40–45.

Nudel, B. (1983). "Consistent-Labeling Problems and Their Algorithms: Expected Complexities and Theory Based Heuristics," *Artificial Intelligence* 21, 135–178.

Schwefel, H.-P. (1995). *Evolution and Optimum Seeking*. Sixth-Generation Computer Technology Series. New York: Wiley.

Selman, B. and H. Kautz. (1993). "Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems." In R. Bajcsy (ed.), *Proceedings of IJCAI'93*. Morgan Kaufmann, pp. 290–295.

Starkweather, T., S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. (1991). "A Comparison of Genetic Sequenceing Operators." In R. Belew and L. Booker (eds.), *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann, pp. 69–76.

Turner, J. (1988). "Almost All *k*-Colorable Graphs are Easy to Color," *Journal of Algorithms* 9, 63–82.

Wolpert, D. and W. Macready. (1997). "No Free Lunch Theorems for Optimization," *IEEE Transactions on Evolutionary Computation* 1(1), 67–82.