# SAW-ing EAs: adapting the fitness function for solving constrained problems

A.E. Eiben
Leiden University
gusz@cs.leidenuniv.nl

J.I. van Hemert
Leiden University
jvhemert@cs.leidenuniv.nl

**Abstract**

In this chapter we describe a problem independent method for treating constraints in an evolutionary algorithm. Technically, this method amounts to changing the definition of the fitness function during a run of an EA, based on feedback from the search process. Obviously, redefining the fitness function means redefining the problem to be solved. On the short term this deceives the algorithm making the fitness values deteriorate, but as experiments clearly indicate, on the long run it is beneficial. We illustrate the power of the method on different constraint satisfaction problems and point out other application areas of this technique.

## 1 Introduction

The common opinion about evolutionary algorithms (EAs) is that they are good optimizers, but cannot handle constraints well. This opinion is based on the observation that the variation operators, mutation and recombination, are 'blind' to constraints. In other words, if the parents satisfy certain constraints the offspring obtained by mutation and/or recombination might violate them. In the last couple of years several options have been proposed to overcome this problem.

Before discussing these options, let us have a closer look on the notion of a constrained problem. A natural classification of problems can be found in [16]. This classification distinguishes free optimization problems, where no constraints are present, and constraint satisfaction and constrained optimization problems that do have constraints to be satisfied. A *free optimization problem* (FOP) is a pair $\langle S, f \rangle$, where $S$ is a free search space (i.e. $S = D_1 \times \ldots \times D_n$ is a Cartesian product of sets) and $f$ is a (real valued)

objective function on $S$, which has to be minimised. A solution of a free optimization problem is an $s \in S$ with an optimal (minimal) $f$-value. A *constrained optimization problem* (COP) is a triple $\langle S, f, \phi \rangle$, where $S$ is a free search space, $f$ is a (real valued) objective function on $S$ and $\phi$ is a formula (Boolean function on $S$). A solution of a constrained optimization problem is an $s \in S$ with $\phi(s) = true$ and an optimal $f$-value. A *constraint satisfaction problem* (CSP) is a pair $\langle S, \phi \rangle$, where $S$ is a free search space and $\phi$ is a formula (Boolean function on $S$). A solution of a constraint satisfaction problem is an $s \in S$ with $\phi(s) = true$. Usually $\phi$ is called the *feasibility condition*, and it is defined by a number of constraints (relations) $c_1, \ldots, c_m$ on the domain, that is the formula $\phi$ is the conjunction of the given constraints. Satisfying the constraints means finding an instantiation of variables $v_1, \ldots, v_n$ within the domains $D_1, \ldots, D_n$ such that the relations $c_1, \ldots, c_m$ hold. Solving a CSP means finding one feasible element of the search space, solving a COP means finding a feasible and optimal element. Solving COPs by EAs is extensively treated in [30, 31, 32, 33] and [34], where different options for constraint handling are given and an experimental comparison of various options can be found. Such surveys or comparative investigations on EAs and CSPs in general are more seldom, at this moment we are only aware of [12] and [16].

Let us note that the problem of handling constraints is present in both COPs and CSPs. For both cases the commonly listed options for treating this problem are the following (after [8, 16, 33]).

1. **Eliminating** infeasible individuals/chromosomes.
2. **Penalizing** infeasible individuals/chromosomes.
3. **Repairing** infeasible individuals/chromosomes.
4. **Special variation operators** preserving the feasibility of the parents.
5. **Special representation/decoding** such that chromosomes always stand for feasible individuals.

It is obvious that options 3, 4 and 5 are problem dependent. In a given problem context they might provide a powerful algorithm, but only little can be said in general about handling constraints this way. Options number 1 and 2 are problem independent, but it is clear that the first one leads to a very inefficient algorithm. Penalizing infeasible individuals/chromosomes has many advantages. First of all, if it is applied to all constraints then minimizing the total penalty is the 'only' thing to be done. In other words, it transforms a COP/CSP into an FOP. Considering that EAs have a 'basic instinct' to

optimize, this is a very natural choice. It is also very transparent. Penalties can be defined independently for each constraint and the total penalty of a chromosome can be the weighted sum of these local penalties. Using weights also allows the user to distinguish between difficult (important) and easy (less important) constraints by giving them a relatively high, respectively low weight. There are, of course, also disadvantages of using penalties. First of all, packing all knowledge on violated constraints into a single number causes a loss of information. Besides, if one is willing to distinguish between constraints, it can be difficult to determine appropriate weights without substantial insight in the problem. Finally, this approach is said not to work in case of sparse problems with only a few solutions [40].

# 2 Determining penalties

Let us summerize the properties of penalty based constraint handling in EAs:

1. Conceptually simple, transparent,
2. Problem independent,
3. Reduces problem to 'simple' optimization,
4. Allows user to tune on his/her preferences by weights,
5. Loss of information by packing everything in a single number,
6. Might require knowledge about the problem (if weights are used),
7. Said not to work well for sparse problems.

Looking carefully at the advantages (items 1 to 4) and disadvantages (5 to 7) of penalty based constraint handling discloses that using appropriate penalties is crucial for the success of this approach. Namely, if the constraints that are more difficult to satisfy have a relatively high weight, then satisfying them gives a relatively high reward to the algorithm. Thus, the EA will be 'more motivated' to satisfy these constraints. For a good performance it is thus essential that the weights reflect the hardness of constraints properly. This causes two difficulties. First, determining the relative hardness of constraints, and thereby the appropriate weights, requires domain knowledge. Second, the definition of appropriate weights can be problem solver dependent — a constraint that is hard for method `A` can be easy for method

B and vice versa. A natural way to handle these problems is to let the problem solver determine the penalties. In case of evolutionary algorithms, this amounts to having the EA determining its own fitness function.

Our first efforts in this direction have been reported in [13], followed by [14, 15]. This approach, called 'learning penalty functions' is based on adjusting the weights of constraints in an off-line fashion, i.e. *after* finishing a run with an EA on a given problem.

*Off-line weight update mechanism*
  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 1: Off-line weight update mechanism

Redefining the fitness function happens by raising the weights of those constraints that are violated by the best individual at termination (thus only in case of unsuccessful runs). Experiments on the so-called Zebra puzzle compare the number of successful runs (out of 100) with and without the learning feature mentioned above. The results turn out to depend on the applied crossover operator, but typically the performance is doubled by using this learning mechanism (in case of one negative outlier the performance does not change, in case of one positive outlier the performance increases by a factor of 6). Inspection of the weights after the whole series of 100 runs with learning exhibits that the weights are to a great extent independent from the applied crossover operator and the initial values of the weights. In Figure 2 we reproduce the curves from [15] showing these outcomes.
These curves show that the learning mechanism is robust, that is insensitive to the specific algorithm setup. This supports the conclusion that the weights learned reflect properties of the problem itself, and are not artifacts of the algorithm or the experimentation.

Subsequent work has been based on the insight that (evolutionary) search is a dynamic process passing different phases. Even though these phases cannot be crisply distinguished, it is widely acknowledged that population dynamics and the corresponding (near) optimal algorithm parameter values are changing during a run. Using the terminology of the beginning of this section one could say that the definition of what appropriate weights are may change during problem solving. Adapting the basic idea of letting the problem solver determine the penalties to this view implies that the weights

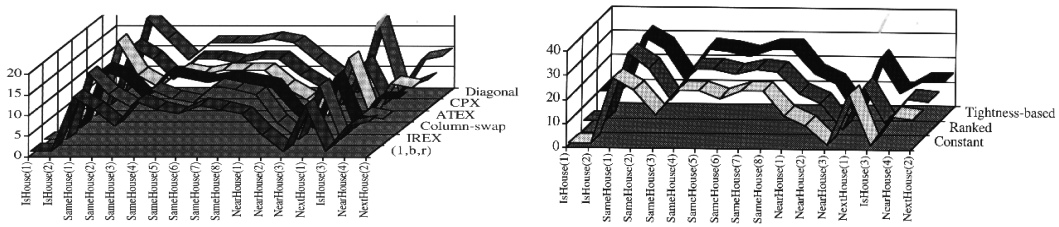


Figure 2: Constraint weights learned on the Zebra puzzle. X-axis: constraints, Z-axis: weights obtained after 100 runs, Y-axis left figure: different crossover operators, Y-axis right figure: different weight initialization methods.

are redefined in an on-line fashion, i.e. *during* a run of the EA on a given problem. The resulting mechanism, called Stepwise Adaptation of Weights (SAW), is presented in Figure 3.

*On-line weight update mechanism*
  set initial weights (thus fitness function $f$)
  **while not** termination **do**
    **for** the next $T_p$ fitness evaluations **do**
      let the GA go with this $f$
    **end for**
    redefine $f$ and recalculate fitness of individuals
  **end while**

Figure 3: On-line weight adaptation mechanism (SAW)

Redefining the fitness function happens by adding a value $\Delta w$ to the weights of those constraints that are violated by the best individual at the end of each period of $T_p$ fitness evaluations. It is clear that the SAW-ing mechanism adds two new parameters to an EA, the length of the update period $T_p$ and the level of weight increase $\Delta w$. Extensive tests on graph colouring and 3-SAT [9, 27] showed that algorithm performance is rather independent from these values, thus they need not to be fine tuned.

# 3 SAW-ing evolutionary algorithms

The SAW-ing mechanism has been applied to various constraint satisfaction problems: graph colouring, satisfiability, and binary CSPs. In this section we briefly summarize the most important results of these studies.

## 3.1 Graph colouring

The first application of SAW-ing concerns graph 3-colouring [9, 11, 17]. The problem of graph 3-colouring is to colour each vertex $v \in V$ of a given undirected graph $G = (V, E)$ with one of three colours so that no two vertices connected by an edge $e \in E$ are coloured with the same colour.

For this problem an order-based EA has been developed where the individuals are permutations of nodes and a decoder constructs a colouring from a permutation. As a decoder a simple greedy algorithm is used which colours a node with the lowest colour[1] that does not violate constraints and leaves nodes uncoloured when this is not possible. Somewhat deviating from the general idea of using the weighted sum of unsatisfied constraints as fitness. Evaluation of a permutation is based on the number of uncoloured nodes in the colouring belonging to it. Formally, the function $f$ is defined as:

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

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

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

Initial weights are set at $w_i \equiv 1$, these weights are increased with a step size of one during updates.

The effect of the SAW-ing mechanism on the EA performance has been tested using swap mutation as the only search operator, a (1+1) selection scheme and the SAW mechanism with $T_p = 250$ and $\Delta w = 1$ on graphs generated with the graph generator written by Joe Culberson[2] using four different seeds. The results on equipartite graphs with $n = 1000$ nodes and $p = 0.010$ edge connectivity are summarized in Table 1. The table shows the results for Falkenauers grouping GA [19, 20], Brélaz' DSatur algorithm with backtracking [4], an EA without SAW-ing, a hybrid EA+DSatur algorithm and the EA with SAW-ing.

These experiments (not all results repeated here) 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 effective, obtaining much better success rates at lower costs, but also very robust.

[1] Colors are represented by integers.

[2] Source code is available at `ftp://ftp.cs.ualberta.ca/pub/joe/GraphGenerator/generate.tar.gz`.

| method | SR | AES |
|---|---|---|
| Grouping GA | 0.00 | 300000 |
| EA | 0.09 | 261221 |
| DSatur | 0.22 | 220033 |
| EA+DSatur | 0.33 | 201354 |
| EA+SAW | 0.92 | 113099 |

Table 1: Success rates (SR) and the average number of evaluations to a solution (AES) for the Grouping GA, (1+1) EA using SWAP, DSatur with backtracking, the hybrid EA+DSatur and the (1+1) SAW-ing EA using SWAP.

A thorough comparison between DSatur with backtracking and a SAW-ing EA is performed on graph instances with three different topologies (arbitrary 3-colourable, equi-partite 3-colourable and flat 3-colourable graphs), three different sizes ($n = 200, 500, 1000$) and for different values of edge connectivities around the phase transition where the hardest instances are located. Globally, the conclusions are that DSatur is better on the 'easy' instances (small graphs, the easier topologies and large graphs far from the phase transition), while the SAW-ing EA is better on the hardest instances. The SAW-ing EA is often able to find solutions where DSatur does not find any. As for speed, in general the EA needs fewer steps.

It is very interesting to see the fitness curve of a run of the EA with the SAW mechanism. Figure 4 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 shows that within each period the fitness (actually, the penalty) drops as the EA is making progress and then sharply rises when the weights are updated, giving the image of a *saw*.

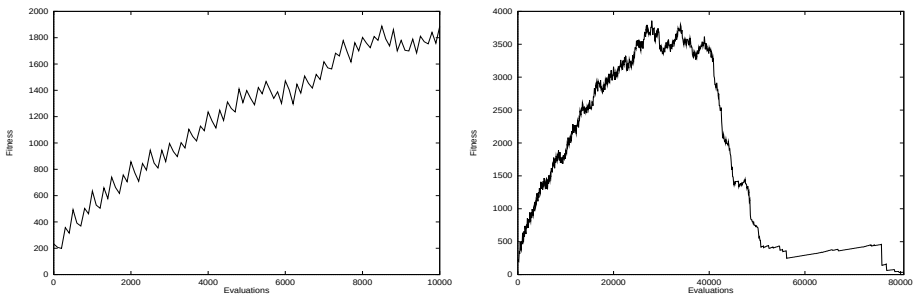


Figure 4: Fitness curve for the SAW-ing EA on graph colouring.

## 3.2 Satisfiability

In a propositional satisfiability problem (SAT) a propositional formula is given and a truth assignment for its variables is sought for that makes the formula true. This problem was the first computational task shown to be NP-hard [5]. Without loss of generality it can be assumed that the given formula is in conjunctive normal form (CNF), i.e. it is a conjunction of clauses where a clause is a disjunction of literals. In the 3-SAT version of this problem it is also assumed that the clauses consist of exactly three literals. In the common notation, a formula has $l$ clauses and $n$ variables. Mitchell *et al.* [35] report that the phase transition, where the hardest problem instances are located, is found when $l = 4.3 \cdot n$.

In [10] and [27] several conclusions on SAW-ing EAs for graph colouring are validated on 3-SAT problems. A straightforward bit-string representation (one literal – one bit) and fitness function (the weighted sum of unsatisfied clauses) in a steady-state style form the basis of the algorithm. Similarily to graph colouring, an EA with population size 1 and mutation only works best for 3-SAT. The relative insensitivity of the SAW-ing mechanism to the parameters $T_p$ and $\Delta w$ is confirmed, and the particular behavior of the fitness function can also be observed. Figure 5, after [27], shows the development of fitness values (the weighted sum of unsatisfied clauses, to be minimized) during a typical run. Although the oscillations are heavier than for graph colouring, the general tendency is similar: the fitness curve is increasing first, then it is suddenly decreasing and hits the optimum level of zero.

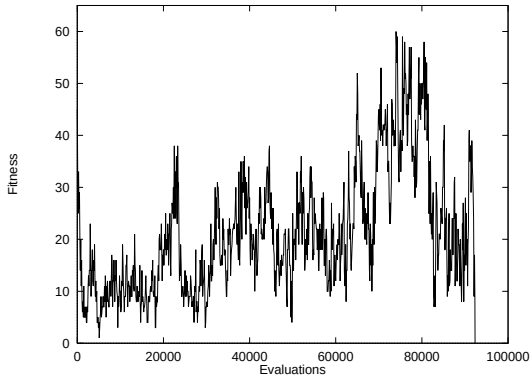


Figure 5: Fitness curve for the SAW-ing EA on 3-SAT.

The paper [10] is concerned with developing a suitable version of the general SAW-ing EA and comparing this algorithm with traditional AI heuristics for solving 3-SAT. The best heuristic algorithms belong to the GenSAT fam-

ily, such as GSAT proposed by Selman *et al.* [42], HSAT by Gent and Walsh [23, 25] that outperformed GSAT, and Frank's WGSAT that was shown to be better than HSAT [22, 24]. The comparison of these techniques on Frank's set of 1000 satisfiable instances (SeedSet1) as well as on 1000 random instances (SeedSet2) show that the SAW-ing EA is superior. The results are given in Table 2 after [10].

| | SeedSet1 | | SeedSet2 | |
|---|---|---|---|---|
| method | SR | AES | SR | AES |
| WGSAT | 0.30 | 110507 | 0.53 | 119262 |
| SAW-ing EA w/o fine tuning | 0.37 | 88359 | 0.68 | 97751 |
| SAW-ing EA with fine tuning | 0.48 | 70182 | 0.88 | 73166 |

Table 2: Comparison between WGSAT and different SAW-ing EAs on 3-SAT.

An interesting aspect of these comparisons is that the family of GenSAT based methods is actually very similar to an asexual EA with extinctive $(1,\lambda)$ selection strategy (that has been used as SAW-ing EA with fine tuning on $\lambda$ in the above experiments). Moreover, the WGSAT algorithm looks very much like a SAW-ing EA, since both methods are using an adaptive weighting mechanism on the clauses to be satisfied. But while WGSAT does an exhaustive search on each neighbour (in EA terms: on each child that can be obtained by one bit-flip mutation), the $(1,\lambda)$ EA only generates $\lambda$ offspring before selecting the next generation (consisting of one single candidate solution). Despite the similarities in the search mechanisms there are significant differences in performance between WGSAT and this SAW-ing EA. These are most probably caused by the fact that the EAs makes locally suboptimal decisions by not performing exhaustive neighbourhood search, while WGSAT enumerates all neighbours (mutants) around a given trial solution and becomes sensitive for local optima. Apparently, the locally suboptimal choices of the EAs prevent getting stuck in local optima and on the long run this leads to a better overall performance.

A standard way of circumventing the local optimum problem of hill-climbers is using them with restarts. In [10] WGSAT is used without restarts in order to keep the differences between the SAW-ing EA and the WGSAT algorithm minimal. Fair as this may seem, this can cause suboptimal performance for WGSAT and thus unfair comparisons. In [1] WGSAT with restarts is considered and the frequency of restarts is fine tuned by numerous tests. Additionally, a second competitor of the SAW-ing EA is added to the contest: an evolution strategy, based on a hint of Z. Michalewicz (personal

communication). The basic idea is to make the originally discrete SAT problem continuous and apply an evolution strategy that has the reputation of a good EA variant in case of continuous variables.

For an experimental comparison solvable problem instances are used created by the generator `mkcnf.c` by Allen van Gelder[3]. To ensure that the problem instances are 'interesting' the ratio $l = 4.3 \cdot n$ is maintained in this investigation. Tests are performed on three instances for each of the four different problem sizes ($n = 30, 40, 50$ and 100). The results obtained by the fine tuned versions of all three algorithms are summarized in Table 3.

| | 30 | | 40 | | 50 | | 100 | |
|---|---|---|---|---|---|---|---|---|
| method | SR | AES | SR | AES | SR | AES | SR | AES |
| ES | 0.31 | 18725 | 0.45 | 10946 | 0.38 | 18068 | 0.15 | 85670 |
| SAW | 1.00 | 34015 | 0.93 | 45272 | 0.85 | 40836 | 0.72 | 50896 |
| WGSAT | 0.99 | 38316 | 0.98 | 31747 | 0.77 | 58386 | 0.36 | 124744 |

Table 3: Summary of the results for ES, SAW-ing EA and WGSAT

Here again the SAW-ing EA is the best algorithm. On the smaller test cases the difference with WGSAT is small. For $n = 40$ WGSAT is slightly better, but the SAW-ing EA clearly scales up better, that is on the largest test cases it has a significantly higher SR and lower AES than WGSAT.

## 3.3 Random binary CSPs

A binary constraint satisfaction problem is a CSP where each constraint is binary, that is concerns exactly two variables. Restricting a study to binary CSPs does not lead to loss of generality, because every CSP can be equivalently transformed into a binary CSP [44]. Binary CSPs have been the subject of research by many others, among which Smith [43] played an important role by trying to estimate the difficulty of CSP instances using four parameters:

1. the number of variables,
2. the domain sizes of these variables,
3. the constraint density of the given problem and
4. the tightness of the constraints.

[3]File available at `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances`

Constraint density is defined as the probability that a constraint exists given two variables; this is a feature of the problem as a whole. Constraint tightness is a measure defined for each individual constraint, being the probability that two values for the variables are in a conflict. Fixing the number of variables and the domain size, the constraint density and the average constraint tightness largely determine the hardness of the problem instances. In Figure 6 we show the landscape of solvability exhibiting the theoretically prediced probability (Z-axis) that an instance has a solution as a function of the constraint density (X-axis) and constraint tightness (Y-axis). In this landscape three different areas can be distinguished. First, the high plateau belonging to low density and tightness values, where the probability of finding a solution is one. Second, the low plateau belonging to high density and tightness values, where the probability of finding a solution is zero. Between these two parts there is a third area of phase transition called the *mushy region* [43], where it is very hard to predict if a particular instance does or does not have a solution.

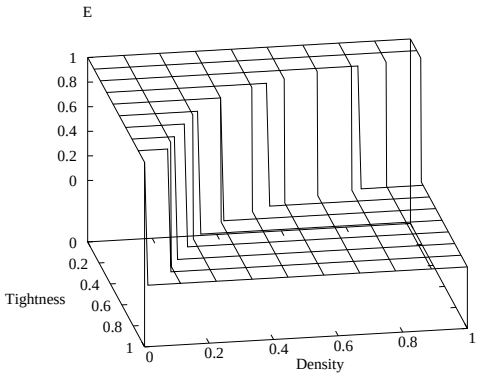


Figure 6: Landscape of solvability as predicted theoretically.

For the experiments on binary CSPs and SAW-ing, a problem instance generator called RandomCsp[4] has been build loosely based G. Dozier's work. The generator is parameterized using the four parameters discussed above. Two series of experiments have been done, an extensive comparison between three adaptive EAs on 25 different density and tightness combinations, and an experiment where only the number of variables is varied to find out the scale-up behaviour of the two best algorithms from the previous experiment.

[4]Available at `http://www.wi.leidenuniv.nl/~jvhemert/csp-ea/`

| | (density,tightness) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | (0.9,0.3) | | (0.5,0.5) | | (0.3,0.7) | | (0.1,0.9) | |
| method | SR | AES | SR | AES | SR | AES | SR | AES |
| MID | 1.00 | 8136 | 0.90 | 26792 | 0.52 | 32412 | 0.96 | 2923 |
| SAW | 1.00 | 3848 | 0.74 | 10722 | 0.23 | 21281 | 0.64 | 1159 |

Table 4: Comparison of MID and SAW. Showing the results for parameter settings that resulted in the biggest difference in performance.

The three methods participating in the first experiment are the coevolutionary GA applied to constraint satisfaction (CCS) [37, 38, 39], the microgenetic method (MID) [3, 6, 7] and the SAW-ing method. The SAW-ing EA used here [12, 28] is the same as the best found in the investigation on graph colouring, using order-based representation with a simple greedy decoder to assign values to variables, a population size of one and one genetic mutation operator that swaps two variables.

For the first series of experiments the number of variables and the domain size of each variable are fixed at 15. Both the density and the tightness values are ranged over $\{0.1, 0.3, 0.5, 0.7, 0.9\}$, resulting in 25 combinations and for each of these combinations 25 instances are generated randomly for the tests. Each algorithm is ran on each instance 10 times and the average success rate and the corresponding AES are recorded for each combination. The success rate results for both MID and SAW show a landscape very similar to the theoretically estimated landscape of solvability. The success rates for CCS drop far sooner than for the other two methods. By the time MID and SAW first have a SR lower than one, CCS is already at SR=0, i.e. not finding any solutions. In Table 4 we reproduce the most interesting results from this experiment, leaving out CCS and showing only the density-tightness combinations from the mushy region. For other combinations MID and SAW almost had the same performance. The results indicate that MID finds more solutions in this region, but SAW is always faster, sometimes even two and a half times as fast as MID.

The second experiment consists of a scale-up test comparing MID and SAW. By fixing the domain size (15), density (0.3) and tightness (0.3) values and varying the number of variables ($n$) from 10 to 40 with a step size of 5, the variance of performance as a function of the problem size can be observed. Figure 7 exhibits the AES results numerically (left) as well as graphically (right). Let us note that the corresponding success rates are constantly 1.0 for for both algorithms. These results show that SAW scales up much better than MID up to $n = 35$, but for $n = 40$ MID is faster, beating SAW by

| $n$ | MID | SAW |
|---|---|---|
| 10 | 10 | 1 |
| 15 | 52 | 2 |
| 20 | 163 | 5 |
| 25 | 410 | 30 |
| 30 | 1039 | 190 |
| 35 | 3462 | 1465 |
| 40 | 17252 | 18668 |

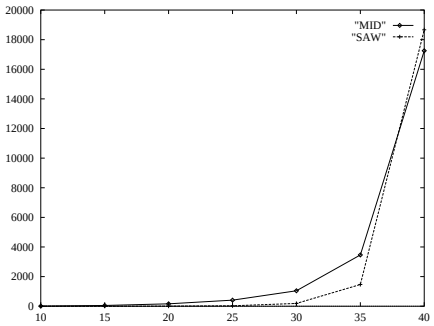


Figure 7: AES results for the scale-up tests of MID and SAW. The density and tightness are set to 0.3 and the domain size of each variable to 15.

almost 1500 fitness evaluations.

Recall the particular behavior of the fitness curves of the SAW-ing EA on graph colouring (Figure 4) and 3-SAT (Figure 5). We have recorded the fitness values during a run on a randomly generated CSP ($n = m = 15$, $d = t = 0.4$) and show the result in Figure 8. Here again, the left curve has a higher resolution, displaying the fitness of the best individual between 0–10000 evaluations, the right curve shows the range 0–100000. The fitness curves show that SAW repeatedly finds local optima resulting in a sudden drop op the fitness. Between these fitness drops SAW shows the same image as it does on the other problems, a periodical rise of the fitness because of an update of the weights that makes the plot look like a saw.

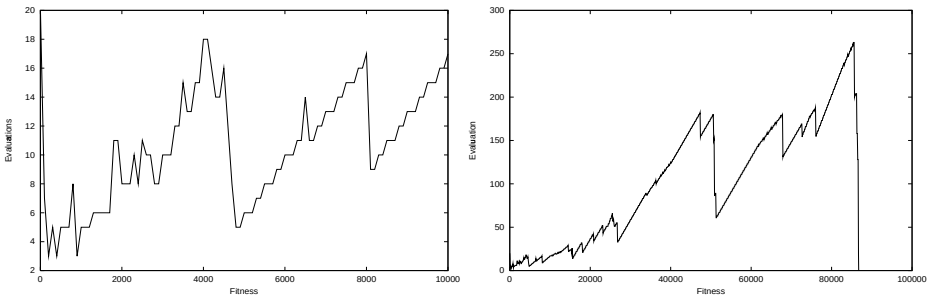


Figure 8: Fitness curves for the SAW-ing EA on binary CSP.

# 4 Related work

The basic idea behind SAW-ing is the re-definition of the function to be optimized, triggered by observed failure in optimizing the current function. The freedom to do so comes from the nature of the objective function representing a CSP. Namely, the objective function must have the property that an optimal function value implies that no constraints are violated. Obviously, there are many functions satisfying this property and as long as this property holds the user is free to choose different variants. It is interesting to note that besides EAs based on penalties other heuristic techniques use a similar idea to solve CSPs by an estimation of the error in a solution that has to be minimized. Also the idea of redefining the heuristic estimation function on-the-fly has been propagated seemingly independently by a number of researchers. In this section we give a brief overview of related work along these lines.

Morris' breakout mechanism is a well-known Artificial Intelligence method in this spirit [36]. The algorithm Morris used in experiments applies an iterative improvement method which proceeds as usual until a local optimum is reached. The function to be optimized is the weighted sum of nogoods, where a nogood is a set of prohibited values. When trapped in a local optimum, the breakout mechanism increases all weights of the current nogoods. When the algorithm breaks out of the local optimum, it resumes with iterative improvement. Good results are obtained on 3-SAT and fair results on graph colouring for three and four colours. Interesting in this paper is the proof that an idealized version of the method will eventually always find a solution for finite CSPs, although this method is not efficient.

In [41], Selman and Kautz describe a GenSAT type method in which GSAT associates a weight with each clause. The weights of all clauses that remain unsatisfied at the end of a *try* are incremented. This very much resembles the off-line weight update mechanism used in the EA for the Zebra problem, see section 2. Recently, Frank adapted this mechanism by updating the weights after each *flip* instead of after each run [21, 22] and achieved very good results with this version called WGSAT. This method is based on the same rationale as our SAW mechanism. Frank envisions his weight adaptation as repeatedly changing the search heuristics. We see SAW-ing as a problem independent way to handle constraints, in evolutionary terms a way to adapt the fitness landscape during the search.

Recent developments in (probabilistic) tabu search, [26], show similarity with this adaptive spirit too. The memory and learning structures described by Løkketangen and Glover embody an adaptive mechanism similar to SAW in the context of surrogate constraint analysis, [29].

There are also a number of techniques within the field of evolutionary computation that modify the definition of fitness during the run. The first one we know of is that of Hadj-Alouane [2], which utilizes feedback from the search process. Technically, the method decreases the penalties for the generation $t + 1$, if all best individuals in the last $k$ generations were feasible, and increases penalties, if all best individuals in the last $k$ generations were infeasible. If there are some feasible and infeasible individuals as best individuals in the last $k$ generations, then the weights remain without change.

We have mentioned the microgenetic algorithm with iterative descent from Dozier *et al.* [3, 6, 7] in Section 3.3. In this EA a small population is used together with a system of breakouts á la Morris. A breakout consists of a nogood (i.e. a pair of values causing a constraint violation) and an associated weight. The standard fitness being the number of violated constraints is extended by the weigthed sum of nogoods as an extra penalty for candidate solutions consisting nogoods. The list of nogoods is created and updated during the run, thus the definition of fitness is also changing during a run. The weight of a breakout is increased every time the pair of values is involved in a constraint violation when the algorithm is in a local optimum. This system is used to make sure the algorithm does not get stuck in local optima.

A different approach to adaptive fitness is represented by coevolutionary methods. This method, also used in the comparison discussed in Section 3.3, is based on two populations in an arms race with each other, as proposed by Paredis [37, 38, 39]. The first population consists of candidate solutions for the given CSP, while the second one contains the constraints. The fitness of an individual is determined by a number of encounters with members of the other population. Individuals are selected for encounters randomly, but biased by their fitness. An encounter is successful (resulting in better fitness) for a solution if it satisfies the encountered constraint. In turn, an encounter is successful for a constraint if the encountered solution cannot satisfy it. This causes the arms race, where the fitness in both populations is continuously varying depending on the (randomized) encounters.

# 5 Concluding remarks

Looking at all research done so far on the SAW-ing method we can highlight the following findings as most important. First of all that a small population size, counterintuitive as it may seem, happens to work very well on the problems that have been tested. Second is the insensitivity that SAW-ing has to its parameters $T_p$ and $\Delta w$. This insensitivity has been found in experiments on graph-colouring and satisfiability. Third, the fitness curves from the three

problem classes shown in Figures 4, 5, and 8 all share the same features and exhibit the shape of a saw. This shape comes from alternating periods of decreasing and increasing fitness values, which are caused by converging to (local) optima and the periodic increase of weights, respectively.

Let us also make a note on the constraint weights the SAW-ing EA finds. The plots of the fitness curves suggest that problem solving with the SAW mechanism happens in two phases. In the first phase the penalty increases a lot because of the increased weights. This is followed by a phase where the penalty drops sharply and hits the optimum. A possible explanation for this behaviour is that in the first phase the EA is learning a good setting for the weights (that is, an appropriate fitness function) thereby making the problem 'easy'. In the second phase the EA is solving the problem, exploiting the knowledge (appropriate weights) learned in the first phase. This interpretation of the fitness curves is a plausible hypothesis. However, suggesting that the EA could learn universally good weights for the given problem instance would go too far. In the first place, another problem solver might need other weights to solve the problem. Besides, we have performed tests on graph colouring and binary CSPs to check this working hypothesis. In particular, we have applied a SAW-ing EA to a problem instance, thus learning a setting of the weights, and then applied an EA to the same problem instance 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. This occurred for both graph colouring and binary CSPs. This refutes the above hypothesis and suggests that the SAW mechanism does not work because it enables the problem solver to discover some hidden, universally good weights. Rather, SAW-ing allows the EA to shift the focus of search (quasi) continuously, and thus amounting to implicit problem decomposition that guides the population through the search space.

Further research on SAW-ing includes other weight update mechanisms and other application areas for this technique. As for the first issue, also decreasing weights instead of only increasing them is a straightforward modifictaion that needs to be assessed. Concerning other application areas, currently we are using genetic programming for data mining where the fitness (to be minimized) is the weighted sum of misclassified cases from the data base.

# References


[1] Th. Bäck, A.E. Eiben, and M.E. Vink. A superior evolutionary algorithm for 3-SAT. In V. William Porto, N. Saravanan, Don Waagen, and A.E. Eiben, editors, *Proceedings of the 7th Annual Conference on Evolutionary Programming*, number 1477 in LNCS, pages 125–136. Springer, Berlin, 1998.

[2] J.C. Bean and A.B. Hadj-Alouane. A dual genetic algorithm for bounded integer programs. Tr-92-53, Department of Industrial and Operations Engineering, The University of Michigan, 1992.

[3] J. Bowen and G. Dozier. Solving constraint satisfaction problems using a genetic/systematic search hybride that realizes when to quit. In Eshelman [18], pages 122–129.

[4] D. Brélaz. New methods to color vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[5] S.A. Cook. The complexity of theorem-proving procedures. In *Proc. of the 3rd Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[6] G. Dozier, J. Bowen, and D. Bahler. Solving small and large constraint satisfaction problems using a heuristic-based microgenetic algorithm. In *Proceedings of the 1st IEEE Conference on Evolutionary Computation*, pages 306–311. IEEE Press, 1994.

[7] G. Dozier, J. Bowen, and D. Bahler. Solving randomly generated constraint satisfaction problems using a micro-evolutionary hybrid that evolves a population of hill-climbers. In *Proceedings of the 2nd IEEE Conference on Evolutionary Computation*, pages 614–619. IEEE Press, 1995.

[8] A.E. Eiben. Constraint satisfaction by evolutionary algorithms, july 1997. Available by `http://www.wi.leidenuniv.nl/~gusz/icga97.ps.gz`.

[9] A.E. Eiben and J.K. van der Hauw. Graph coloring with adaptive evolutionary algorithms. Technical Report TR-96-11, Leiden University, August 1996. Also available as `http://www.wi.leidenuniv.nl/~gusz/graphcol.ps.gz`.

[10] A.E. Eiben and J.K. van der Hauw. Solving 3-SAT with adaptive Genetic Algorithms. In *Proceedings of the 4th IEEE Conference on Evolutionary Computation*, pages 81–86. IEEE Press, 1997.

[11] A.E. Eiben, J.K. van der Hauw, and J.I. van Hemert. Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4(1):25–46, 1998.

[12] A.E. Eiben, J.I. van Hemert, E. Marchiori, and A.G. Steenbeek. Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In A.E. Eiben, Th. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Proceedings of the 5th Conference on Parallel Problem Solving from Nature*, number 1498 in LNCS, pages 196–205, Berlin, 1998. Springer.

[13] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. Repairing, adding constraints and learning as a means of improving GA performance on CSPs. In J.C. Bioch and S.H. Nienhuiys-Cheng, editors, *Proceedings of the 4th Belgian-Dutch Conference on Machine Learning*, number 94-05 in EUR-CS, pages 112–123. Erasmus University Press, 1994.

[14] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. Constrained problems. In L. Chambers, editor, *Practical Handbook of Genetic Algorithms*, pages 307–365. CRC Press, 1995.

[15] A.E. Eiben and Zs. Ruttkay. Self-adaptivity for constraint satisfaction: Learning penalty functions. In *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*, pages 258–261. IEEE Press, 1996.

[16] A.E. Eiben and Zs. Ruttkay. Constraint satisfaction problems. In Th. Bäck, D. Fogel, and M. Michalewicz, editors, *Handbook of Evolutionary Algorithms*, pages C5.7:1–C5.7:8. IOP Publishing Ltd. and Oxford University Press, 1997.

[17] A.E. Eiben and J.K. van der Hauw. Adaptive penalties for evolutionary graph-coloring. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution'97*, number 1363 in LNCS, pages 95–106. Springer, Berlin, 1997.

[18] L.J. Eshelman, editor. *Proceedings of the 6th International Conference on Genetic Algorithms*. Morgan Kaufmann, 1995.

[19] E. Falkenauer. A new representation and operators for genetic algorithms applied to grouping problems. *Evolutionary Computation*, 2(2):123–144, 1994.

[20] E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30, 1996.

[21] J. Frank. Learning short-term weights for GSAT. Technical report, University of California at Davis, October 1996. available as `http://rainier.cs.ucdavis.edu/\verb+~+frank/decay.ml96.ps`.

[22] J. Frank. Weighting for Godot: Learning heuristics for GSAT. In *Proceedings of the AAAI*, pages 338–343, 1996.

[23] I. Gent and T. Walsh. The enigma of SAT hill-climbing procedures. Technical Report 605, Department of AI, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland, 1992.

[24] I. Gent and T. Walsh. Unsatisfied variables in local search. In J. Hallam, editor, *Hybrid Problems, Hybrid Solutions*. IOS Press, 1995.

[25] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing prodedures for SAT. *AAAI*, pages 28–33, 1993.

[26] F. Glover. Tabu search and adaptive memory programming — advances, applications, and challenges. In R.S. Barr, R.V. Helgason, and J.L. Kennington, editors, *Interfaces in Computer Science and Operations Research*, pages 1–75. Kluwer Academic Publishers, Norwell, MA, 1996.

[27] J.K. van der Hauw. Evaluating and improving steady state evolutionary algorithms on constraint satisfaction problems. Master's thesis, Leiden University, 1996. Also available as `http://www.wi.leidenuniv.nl/MScThesis/IR96-21.html`.

[28] J.I. van Hemert. Applying adaptive evolutionary algorithms to hard problems. Master's thesis, Leiden University, 1998. Also available as `http://www.wi.leidenuniv.nl/~jvhemert/publications/IR-98-19.ps.gz`.

[29] A. Løkketangen and F. Glover. Surrogate constraint methods with simple learning for satisfiability problems. In D.-Z. Du, J. Gu, and P. Pardolos, editors, *Proceedings of the DIMACS workshop on Satisfiability Problems: Theory and Applications*, 1996. to appear.

[30] Z. Michalewicz. Genetic algorithms, numerical optimization, and constraints. In Eshelman [18], pages 151–158.

[31] Z. Michalewicz. A survey of constraint handling techniques in evolutionary computation methods. In J.R. McDonnell, R.G. Reynolds, and D.B. Fogel, editors, *Proceedings of the 4th Annual Conference on Evolutionary Programming*, pages 135–155. MIT Press, 1995.

[32] Z. Michalewicz and N. Attia. Evolutionary optimization of constrained problems. In A.V. Sebald and L.J. Fogel, editors, *Proceedings of the 3rd Annual Conference on Evolutionary Programming*, pages 98–108. World Scientific, 1994.

[33] Z. Michalewicz and M. Michalewicz. Pro-life versus pro-choice strategies in evolutionary computation techniques. In Palaniswami M., Attikiouzel Y., Marks R.J., Fogel D., and Fukuda T., editors, *Computational Intelligence: A Dynamic System Perspective*, pages 137–151. IEEE Press, 1995.

[34] Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4(1):1–32, 1996.

[35] D. Mitchell, B. Selman, and H.J. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the AAAI*, pages 459–465, San Jose, CA, 1992.

[36] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the 11th National Conference on Artificial Intelligence, AAAI-93*, pages 40–45. AAAI Press/The MIT Press, 1993.

[37] J. Paredis. Co-evolutionary constraint satisfaction. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, number 866 in Lecture Notes in Computer Science, pages 46–56. Springer-Verlag, 1994.

[38] J. Paredis. Co-evolutionary computation. *Artificial Life*, 2(4):355–375, 1995.

[39] J. Paredis. Coevolving cellular automata: Be aware of the red queen. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA, 1997. Morgan Kaufmann.

[40] J.T. Richardson, M.R. Palmer, G. Liepins, and M. Hilliard. Some guidelines for genetic algorithms with penalty functions. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 191–197. Morgan Kaufmann, 1989.

[41] B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of IJCAI*, 1993.

[42] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 440–446, 1992.

[43] B.M. Smith. Phase transition and the mushy region in constraint satisfaction problems. In A. G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 100–104. Wiley, 1994.

[44] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.