

A Conflict Based SAW Method for Constraint Satisfaction Problems

Rafi Shalom, Mireille Avigal, and Ron Unger

Abstract—Evolutionary algorithms have employed the SAW (Stepwise Adaptation of Weights) method in order to solve CSPs (Constraint Satisfaction Problems). This method originated in hill-climbing algorithms used to solve instances of 3-SAT by adapting a weight for each clause. Originally, adaptation of weights for solving CSPs was done by assigning a weight for each variable or each constraint. Here we investigate a SAW method which assigns a weight for each *conflict*. Two simple stochastic CSP solvers are presented. For both we show that constraint based SAW and conflict based SAW perform equally on easy CSP samples, but the conflict based SAW outperforms the constraint based SAW when applied to hard CSPs. Moreover, the best of the two suggested algorithms in its conflict based SAW version performs better than the best known evolutionary algorithm for CSPs that uses weight adaptation, and even better than the best known evolutionary algorithm for CSPs in general.

I. INTRODUCTION

Weight adaptation originated in the context of 3-SAT solvers. In [1] Selman and Kautz introduced weight adaptation as a way to enhance GSAT [2], which is a hill-climber with restart algorithm for the 3-SAT problem. Each clause of the CNF has a weight associated with it, and the algorithm tries to minimize the sum of the weights of unsatisfied clauses rather than minimizing their number. Weights of unsatisfied clauses are increased before each restart. This approach was later improved by Frank [3] by changing the weights at every search step, namely whenever a variable value is flipped. When used for Genetic algorithms (GAs), and evolutionary algorithms (EAs), adaptation of weights was termed SAW (Stepwise Adaptation of Weights). The SAW technique was used to enhance EAs solving a variety of constraint satisfaction problems [4], [5] including 3-SAT [6], and general CSPs [7]. The weights were either weights per constraint, or per variable. The search space was represented either by an array of values, one for each variable, or by a permutation of the variables, decoded into an assignment to some of the variables by a special decoder [5]. Weights were updated at equal intervals according to the constraints violated by the best individual of the population, or according to its variables that could not be assigned by the decoder.

In [7], [8] it was concluded that using a weight per variable vs. a weight per constraint does not make a significant differ-

ence in the performance of SAW, countering the intuition that keeping more weights, and thus having more information, should enhance the performance of SAW. Here we explore the possibility of a weight adaptation scheme for CSPs that keeps a weight for each conflict. This means that instead of keeping a weight for each variable or each constraint, we keep a weight for all possible “illegal” assignments of values to two variables. We show that keeping weights that are sensitive not only to violated variables but also to their values, significantly improves the performance of SAWing EAs solving constraint satisfaction problems.

Conflict based SAW is suggested especially for CSP solvers, because for other classes of problems for which SAW was proposed we can expect only a slight difference with previous suggestions, if at all. When weight adaptation is used for 3-SAT the algorithms keep a weight for each clause of the CNF. Thus an adaptable weight exists for each set of three variables and their values that render the CNF unsatisfied. This is equivalent to keeping a weight for each conflict. This means that keeping a weight for each conflict is a somewhat traditional approach to weight adaptation. The difference between a conflict based SAW and proposed algorithms for graph 3-coloring does exist though quite small. Although a weight per variable approach was advocated, a version with a weight per graph edge was checked as well and found to be less suitable [5]. The parallel of a conflict based SAW in this case means keeping a weight for any combination of a graph edge and the color the adjacent nodes connected by the edge have when a violation occurs. This only multiplies the number of weights by three w.r.t. the version that keeps a weight per edge, which is not expected to make a big difference. The situation is totally different when it comes to SAWing algorithms for CSPs. The number of conflicts usually exceed the number of constraints and variables by several orders of magnitude, depending on the domain sizes of the variables and the particular CSP.

We introduce two different EAs and apply conflict based SAW and constraint based SAW to both. The key parameter of SAW that determines the interval between two weight adaptations is checked for robustness. Its impact on both weight adaptation schemes is also checked to make sure that using the same value for both schemes does not compromise the comparison. We also provide an estimation for reasonable values of this parameter, which we use in our experiments. In order to determine the impact of conflict based SAW, the conflict based SAW variants of the algorithms are compared with the constraint based SAW variants, on problems of varying hardness. A comparison of the best of the two

R. Shalom is with the Computer Science Division, The Open University of Israel, 108 Ravutski St., Raanana 43107, Israel. (email : rafi.shalom@gmail.com).

M. Avigal is with the Computer Science Division, The Open University of Israel, 108 Ravutski St., Raanana 43107, Israel. (email : miray@openu.ac.il).

R. Unger is with the Mina & Everard Goodman Faculty of Life Sciences, Bar-Ilan University, Ramat Gan, 52900, Israel. (email : ron@biomodel.os.biu.ac.il).

conflict based SAW variants with other EAs for CSPs is also conducted.

The paper is organized as follows : In section II we describe the basic concepts, and detail the algorithms we use, including the constraint based SAW and the conflict based SAW methods. We also describe the test-sets, and the performance measures used. In section III we deal with parameter issues. In section IV we compare the results of a conflict based SAW algorithm with results of recently proposed EAs for CSPs. In section V we explore the effectiveness of conflict based SAW for CSPs of varying hardness. The conclusions are presented in section VI.

II. BASIC CONCEPTS, ALGORITHMS AND MEASURES

A. Binary Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) consists of a finite set $Z = \{X_1, \dots, X_n\}$ of n variables with respective domains D_1, \dots, D_n , and a set of constraints C . For $2 \leq k \leq n$ a constraint $R_{i_1, i_2, \dots, i_k} \in C$ is a subset of $D_{i_1} \times D_{i_2} \times \dots \times D_{i_k}$, where i_1, i_2, \dots, i_k are distinct. R_{i_1, i_2, \dots, i_k} is called a constraint of arity k that bounds the variables X_{i_1}, \dots, X_{i_k} . There are no two constraints that bound the same variables. R_{i_1, i_2, \dots, i_k} is called *restrictive* when $R_{i_1, i_2, \dots, i_k} \neq D_{i_1} \times D_{i_2} \times \dots \times D_{i_k}$. Here we use the term constraint only for restrictive constraints.

An n -tuple (d_1, \dots, d_n) such that $d_i \in D_i$ satisfies R_{i_1, i_2, \dots, i_k} if $(d_{i_1}, \dots, d_{i_k}) \in R_{i_1, i_2, \dots, i_k}$, otherwise it violates R_{i_1, i_2, \dots, i_k} . A solution of a CSP is an n -tuple (d_1, \dots, d_n) which satisfies all constraints.

A *binary constraint satisfaction problem* is a CSP for which all constraints are of arity two. In this paper we limit our attention to binary CSPs, known to be equivalent to general CSPs [9]. Moreover, we discuss only problems for which all variables have the same domain size. We use m to denote the domain size of all variables in our CSPs.

A *conflict* is an assignment of values to variables, which violates a constraint. We use the term conflict only for assignments of values to two variables. A *conflict check* (CC) is a basic operation performed by binary CSP solvers. Given two variables and their values, a conflict check determines in constant time whether or not there is a conflict.

Density and *tightness* are two parameters that measure how constrained a given instance of a CSP is. The density parameter, denoted by p_1 , is the ratio between the number of constraints and the maximum number of possible constraints. Accordingly, the number of constraints in a binary CSP is $p_1 \binom{n}{2}$. The tightness of a single constraint R_{i_1, i_2} is the ratio between the number of conflicts that involve variables X_{i_1}, X_{i_2} and the number of possible assignments to those variables, namely, $1 - \frac{|R_{i_1, i_2}|}{|D_{i_1} \times D_{i_2}|}$. The average tightness (or simply tightness) of a CSP, denoted by p_2 , is the average tightness of all (restrictive) constraints.

B. Weight Adaptation Methods

Weight adaptation was found useful for improving local search algorithms including 3-SAT solvers such as GSAT [1],

[10], [3], and EAs designed to solve a variety of problems [5], [4], [6]. Being a general method, weight adaptation can be customized for a specific problem solver. It is especially easy to apply to local search algorithms, genetic algorithms and evolutionary algorithms. The basic principle of weight adaptation is that instead of minimizing the number of violated constraints (in CSPs), or the number of unsatisfied clauses (in 3-SAT), etc. we keep an adaptable weight for each constraint (or clause), and minimize their sum. Since these weights are always kept positive, weight based objective functions reach a minimum value of zero for, and only for, solutions. Increasing the weights of “problematic” constraints or clauses helps the algorithm reject previously unsuccessful search paths, hopefully escaping local optima. In time, the weight adaptation heuristic is supposed to allow the algorithm to learn which constraints or clauses are harder to satisfy, and rate points in the search space accordingly.

A more formal, yet still general, description of weight adaptation follows. A weight adaptation heuristic $WA(S, V, \chi; \Delta w, T_p)$ consists of a finite search space S , finite set V of *violation entities*, and a function $\chi : S \times V \rightarrow \{0, 1\}$ such that for each $s \in S$ and each $i \in V$, $\chi(s, i) = 1$ iff s violates i . T_p determines the intervals between weight adaptations. Whenever weights are revised they are incremented by Δw .

The algorithm keeps an adaptable weight w_i for each $i \in V$. The objective function to be minimized is $f(s) = \sum_{i \in V} \chi(s, i) w_i$. All weights are initialized to 1. After the creation and evaluation of each new T_p search points, weights are adapted. This is done by letting s' be the latest produced search point, or in the case of an EA the best search point in the latest population, and performing $w_i \leftarrow w_i + \chi(s', i) \Delta w$ for all $i \in V$. This simple framework covers most of the weight adaptation schemes for CSPs.

When the above approach is used for 3-SAT [6], [3], [1], V is the set of clauses, S is all possible assignments of Boolean values to the variables, and an assignment violates a clause if the clause is unsatisfied by the assignment.

For CSPs and graph coloring there are two common approaches. In the first approach S represent all possible assignments to variables, and V is the set of constraints. The other approach is to let S be all the permutations of the variables, and let V be the set of variables. A greedy decoder is used to assign values to the variables in the order that the permutation dictates, by assigning the first value consistent with previous assignments. Variables that cannot be assigned consistently with previous ones are considered violated. Though it is less straightforward, the permutation approach is widely used, because it produced better results than the first approach. The original SAW algorithm for the graph 3-coloring problem used a permutation representation, and rSAWEA (Stepwise-Adaptation-of-Weights EA with randomly initialized domain sets) [11], [12], [13], which is the best known SAWing algorithm for CSPs, uses it as well. This approach was also found to be superior under extensive experimentation [8].

There are a few exceptions which are not covered by the above weight adaptation paradigm, namely refinement, decay, and weight exponentiation originally suggested for 3-SAT solvers [10], [3], [14] which will not be surveyed here. Refinement and decay were found not helpful for CSPs when implemented both with variable based SAW and with constraint based SAW [7]. Other weight adaptation techniques confined to 3-SAT hill-climbers, will not be discussed either. References and implementations of up to date hill-climbers for 3-SAT, including ones using weights, can be found at <http://www.satlib.org/>.

In order to demonstrate that by using a weight for each conflict weight adaptation for CSPs becomes significantly more effective, we compare two weight adaptation schemes. Both use all possible assignments to variables as the search space. For the first, which we use as a constraint based SAW representative we use the set of constraints for the set of violation entities V . For the other which we term CSAW we use the set of all conflicts for V . In section II-C we introduce two simple EAs, and two weight adaptation variants for each. By adding “SAW” and “CSAW” at the end of the names of the proposed algorithms we declare the application of constraint based SAW, and conflict based SAW respectively to each of the algorithms. The choice of Δw and T_p is discussed in section III.

C. The Proposed Algorithms

A $(\mu + \lambda)$ EA is an EA that keeps a population of size μ , and generates additional λ individuals at each new generation. μ individuals of the $\mu + \lambda$ individuals are selected to become the population of the next generation. In a (μ, λ) EA a new generation of size μ is selected only from the λ individuals generated.

In order to compare SAW with CSAW we use two algorithms. The first is a $(1 + 1)$ EA which at each iteration may or may not replace the current search point with a randomly chosen search point that differs only by the value of one variable. The other is a stochastic hill-climber, which at each iteration optimizes one randomly chosen variable by checking all possible assignments to it (keeping other variables fixed), and thus could be seen as a $(1 + (m - 1))$ EA. The merits of using $(1 + \lambda)$ EAs are that they are very simple algorithms. Moreover, $(1 + 1)$ and $(1, \lambda)$ EAs were traditionally used with SAW [6], [5].

Though keeping a population of size 1 may seem problematic in the context of genetic algorithms, it was argued upon the introduction of SAW [5] that a $(1 + 1)$ algorithm may be viewed as a GA with an extreme value for the population size parameter, and that either way it fits the broader notion of an evolutionary algorithm. Regardless of how $(1 + \lambda)$ algorithms are classified, they provide a convenient basis for a comparison of SAW with CSAW.

We turn to a detailed description of the algorithms. Both algorithms require the following :

- A function that performs a conflict check in constant time, and increases the conflict checks counter. The

function receives two variables and their values and returns true if there is a conflict and false otherwise.

- A list of pairs of variables bounded by constraints, one pair of variables per constraint.
- For each variable, a list of variables that share a constraint with the variable.
- A parameter for the maximum number of conflict checks the algorithm may use for the search, called *maxCC*.

The $(1 + 1)$ algorithm without weight adaptation simply chooses an initial assignment to all variables, and at each iteration mutates the only individual by uniformly selecting a variable and changing its value to a uniformly selected value different than its current value. Selection is done by evaluating the current search point and the mutated search point, and preferring the later if it evaluates at least as good as the former. The algorithm keeps running as long as no solution is found and the conflict checks counter is less than *maxCC*.

The hill-climbing algorithm, which we term *HC*, works in a similar way. The only difference is that instead of checking only one value of the selected variable at each iteration, the search points with all values of the variable are evaluated, and the best one is selected. The search point with the current value of the variable is evaluated first, and then the search points with other values of the variable are evaluated according to the order of the values in a static domain (any domain order may be used without special consideration). Whenever a search point which is at least as good as the best one thus far is encountered, it is taken to be the best one yet. Since we always select one individual from m individuals that include the parent population of size 1, this is a $(1 + (m - 1))$ EA.

When optimizing a variable in the HC algorithm, it is possible that one of its values does not conflict with the values of other variables. Since no other value of the optimized variable can produce better results, we avoid the evaluation of the rest of the values. For the purpose of weight adaptation we still assume that all the $m - 1$ individuals were created.

Each of the two algorithms has two weight adaptation variants. Adding a weight adaptation scheme requires the parameters Δw , T_p , and a table for the weights. Weight adaptation variants that use a weight per constraint is called a SAW variant, and one that uses a weight per conflict is called a CSAW variant. We use $(1 + 1)$ SAW and $(1 + 1)$ CSAW to denote the $(1 + 1)$ algorithms, and HCSAW, HCCSAW to denote the hill-climbing algorithms. In a SAW type algorithm a weight is determined by the violated variables, while in a CSAW type algorithm the values of the variables are also used in order to read and update weights. Otherwise the SAW and CSAW variants of the same algorithm are exactly the same.

In order to keep a weight for each conflict while achieving optimal access time, the CSAW algorithms keeps a four dimensional table with an entry for all possible assignments to any two variables. The SAW algorithms need only a two

dimensional matrix with an entry for any pair of variables. However, CSPs and especially random CSPs usually use a four dimensional matrix of Boolean values to perform efficient conflict checks, and a CSAW algorithm can simply use such a matrix with two bytes per entry that keeps integer weight values instead of a Booleans. Either way, the costs of holding and setting up a four dimensional matrix are polynomial and remain virtually unnoticed even for relatively large CSPs. If the direct access table becomes too large to fit in main memory, a hash table of conflicts is a reasonable alternative.

Weight adaptation is done once each new T_p search points are created and evaluated. It is performed using the current search point by going over the list of all constraints. For each pair of variables bounded by a constraint, the relevant weight is increased by Δw if there is a conflict involving the two variables. For simplicity we assume that the HC algorithm creates exactly $m - 1$ search points at each iteration, and always use T_p values divisible by $m - 1$.

The straightforward way to evaluate an individual would be to go over the list of all the pairs of variables bounded by a constraint, and summing the weights for which the current assignment to the pair of variables constitutes a conflict. This also provides a way to determine weather the evaluated search point is a solution. A more efficient way is to avoid a full evaluation and use the fact that we only need to compare search points that differ by the value of one variable. Going over the list of all constraints that bound that particular variable and summing the weights relevant to it is sufficient because all costs that do not involve it remain the same.

In order to track the (total) number of violated constraints, thus keeping the ability to detect solutions, we keep a variable called $Tvio$, which holds the number of violated constraints for each current search point. We compute the total number of constraint violations only once for the initial search point and store it in $Tvio$. At every iteration, we advance from search point s_1 to s_2 , by replacing the value of a single variable X . Let $CV_{s,X}$ denote the number of constraint violations in s of constraints that bound X . The values of $CV_{s_1,X}, CV_{s_2,X}$ are computed while summing the weights relevant to X in s_1 and s_2 , by counting constraint violations that involve X (which is the same as the number of added weights). $Tvio$ can now be updated to hold the number of constraint violations of s_2 by letting $Tvio \leftarrow Tvio - CV_{s_1,X} + CV_{s_2,X}$. Knowing the total number of constraint violations for all search points of the algorithm may also be useful if one wants a version of the algorithm that outputs the best search point the algorithm produced in case a solution was not found.

The HC algorithms can be improved by choosing the variable to be “hill-climbed” from variables different than the one in the previous iteration, because there is no merit in optimizing the same variable in succession. Moreover, both in the HC algorithms and the $(1 + 1)$ algorithms, one can avoid the re-evaluation of the selected individual in the following iteration. This would require more complicated

implementation, and additional data structures. Since the SAW and CSAW versions of the algorithms differ only in the application of weights, the above changes do not affect the comparison of SAW with CSAW, and they are avoided. Finding more efficient algorithms and supportive data structures is left for further research.

D. Test-sets

An algorithm that guarantees a solution if one exists, and able to identify the problem as insoluble is called a *complete algorithm*. The time needed for complete algorithms such as backtracking algorithms [15] to solve a CSP indicates how hard the CSP is. CSP solvers are frequently tested with instances of random binary CSPs. It is possible to use the density and tightness parameters in order to produce CSPs of varying hardness.

Binary CSPs may be classified using four parameters : n (the number of variables), m (the domain size of each variable), p_1 (density), and p_2 (tightness). Keeping n, m , and p_1 constant, small values of p_2 produce binary CSPs with many solutions, thus easily solved ones. Large values of p_2 create insoluble CSPs, and the higher p_2 is makes it easier for complete algorithms to recognize CSPs as insoluble. Given specific values of n, m , and p_1 , there are values of p_2 for which CSPs have very few solutions, or are insoluble in a way that is hard to recognize. Around these parameter values there is a sharp peak in the time required by complete algorithms. This phenomenon is sometimes referred to as the *phase transition phenomenon*. Note that unlike backtracking algorithms, local search algorithms are not able to recognize insoluble CSPs, and they are therefore usually tested only with soluble CSPs.

Three test-sets of randomly generated CSPs from three different models are used to measure performance. For a comparison of the proposed CSAW algorithms with other EAs for CSPs we use a model F [16] test-set of soluble phase transition CSPs with 10 variables of domain size 10. It consists of nine categories, denoted by $F-1, F-2, \dots, F-9$, each holding 25 problems. The categories have the following density-tightness combinations $F-1:(0.1,0.9)$, $F-2:(0.2,0.9)$, $F-3:(0.3,0.8)$, $F-4:(0.4,0.7)$, $F-5:(0.5,0.7)$, $F-6:(0.6,0.6)$, $F-7:(0.7,0.5)$, $F-8:(0.8,0.5)$, $F-9:(0.9,0.4)$. The test-set was created by Craenen [11], and is downloadable at http://www.emergentcomputing.org/csp/testset_mushy.zip.

Another phase transition test-set is an RB model [17] test-set that has five instances of soluble phase transition CSPs. The CSPs are random binary CSPs with 30 variables of domain size 15. We use this test-set to check the influence of the T_p parameter over larger problems, and as another source for phase transition CSPs. The test-set was created for the annual SAT competition and is downloadable at <http://www.nlsde.buaa.edu.cn/kexu/benchmarks/frb30-15-cnftar.gz>.

The third test-set contains a model E [18] test-set created by Craenen, and used in [7], [8]. Model E employs only one parameter instead of a density-tightness combination to control the number of conflicts in the generated random

CSPs. Given a parameter $0 \leq p \leq 1$ a model E CSP is created by choosing uniformly, independently and with repetitions $p \binom{n}{2} m^2$ conflicts. The test-set contains ten categories, denoted by $E-1, E-2, \dots, E-10$. Each category contains 25 instances of soluble CSPs with 15 variables, each having a domain of 15 values. p ranges from 0.2 in the first category to 0.38 in the last, increasing by 0.02. In other words, category $1 \leq i \leq 10$ has problems for which $p = 0.2 + 0.02 \cdot (i - 1)$. This means that the first category contains very easy CSPs, and that we approach the phase transition as we get closer to the last category. Note that choosing from all possible conflicts means that even for very small values of p it is almost certain that all pairs of variables become constrained, so we expect a density of 1 in all problems in this test-set. The test-set is downloadable at http://www.xs4all.nl/braenen/resources/csp_modelE.v15.d15.tar.gz.

E. Performance Measures

Each CSP instance testing any algorithm is always run 50 times. The first and most important measure is SR (success rate) which measures the percentage of runs at which a solution was found. Note that since only soluble instances were used, a solution always exists. A secondary measure which is supposed to measure the runtime needed to find a solution is ACCS (Average Conflicts Checks to Solution). It averages the number of conflict checks in successful runs. If there are no successful runs ACCS is not defined.

It was customary to use AES (Average Evaluations to Solution), which measures the number of evaluations of search points in successful runs. ACCS is now preferred, mainly because AES is not able to measure some of the costs (such as the time needed to run the decoder in a permutation representation SAW algorithm, and the costs needed for weight adaptation). Using conflict checks to measure performance is also the standard for backtracking algorithms [15]. For the purpose of comparing the proposed SAW and CSAW algorithm variants the merits of AES and ACCS are about the same. ACCS is preferred here because it was used to measure recent relevant EAs, and because it is finer. For the algorithms suggested here it is also strictly runtime proportional.

III. TESTING PARAMETER INFLUENCE

As was mentioned in section II-B, weight adaptation requires the parameters Δw and T_p . Experiments [5], [6] show that weight adaptation is a robust technique. Hardly any difference is noticed when using different Δw values, and the usual choice of $\Delta w = 1$ seems safe enough. T_p shows robust behavior provided it has a large enough value, and the traditional value $T_p = 250$ was used many times without special consideration.

We devote special attention to T_p for two reasons. The first reason is that we want to make sure that using the same T_p value does not bias the comparison. Namely, that the SAW and CSAW variants of an algorithm do not have different performance peaks for significantly different T_p values. The other reason is that it is plausible that problems with a higher

number of variables or with larger domains would require longer periods between weight adaptations.

In order to use T_p values that take problem size into consideration we define $C = n(m - 1)$ as the *cover value* for a CSP. This value is the total number of search points obtainable by changing the value of a single variable. For all test-sets, tests are conducted for T_p values from $0.2C$ to $5.8C$ increasing by $0.4C$. In other words, all tests are done 15 times, with $T_p = \beta C$ where $\beta = 0.2 + 0.4i$, and $0 \leq i \leq 14$. Since HC is assumed to create $m - 1$ individuals at each iteration, the $(1 + 1)$ variants perform a weight adaptation once every $\beta n(m - 1)$ iterations, while the HC variants do this only once every βn iterations.

Figure 1 shows the results for running $(1 + 1)$ and HC with both SAW and CSAW for five test-set categories, each with one of the two measures. Model E and model F results are for $maxCC = 1,000,000$ and model RB results are for $maxCC = 30,000,000$. Note that the ACCS measure applies only to successful runs. In most cases all the runs were successful thus ACCS reflects the average of all runs. However in the model F comparisons only the CSAW algorithms had a 100% success rate. The SAW algorithms did not always find a solution, but all had over 80% success.

As expected, lower T_p values hurt performance, and higher T_p values usually do not hurt performance substantially, if at all. CSAW algorithms are more sensitive to the parameter, especially when T_p is set to lower values. Except for low T_p values any choice of the parameter seems to allow a fair comparison between SAW and CSAW. The value $T_p = 1.4C$, shown on the forth column from the left of all the graphs in figure 1, is chosen for our experiments. Though $T_p = 1.4C$ is a reasonable choice, in case of doubt higher values such as $T_p = 2C$ and even higher may be used without fear of significantly hurting performance. CSAW outperforms SAW at all tests performed for model F and model RB, which indicates that CSAW consistently outperforms SAW on phase transition CSPs.

Results for the RB model show that the traditional value $T_p = 250$ may be too low for CSAW as problem size increases. This justifies a choice of T_p that takes problem size into account.

The HC algorithms perform about twice as fast as the parallel $(1 + 1)$ algorithms over all test-sets. Since HCCSAW is the best of the two CSAW algorithms, it is used for a comparison of CSAW algorithms with other EAs for CSPs.

IV. COMPARING HCCSAW WITH VARIOUS EAS FOR CSPS

In order to evaluate the effect CSAW has when applied to a simple stochastic algorithm, model F test-set results for HCSAW and HCCSAW are compared. This test-set has the advantage that many relevant algorithms were checked using it, and the results were published [11], [12], [13]. The most relevant comparison is of HCCSAW with rSAWEA (Stepwise-Adaptation-of-Weights EA with randomly initialized domain sets), which is the best known SAWing EA.

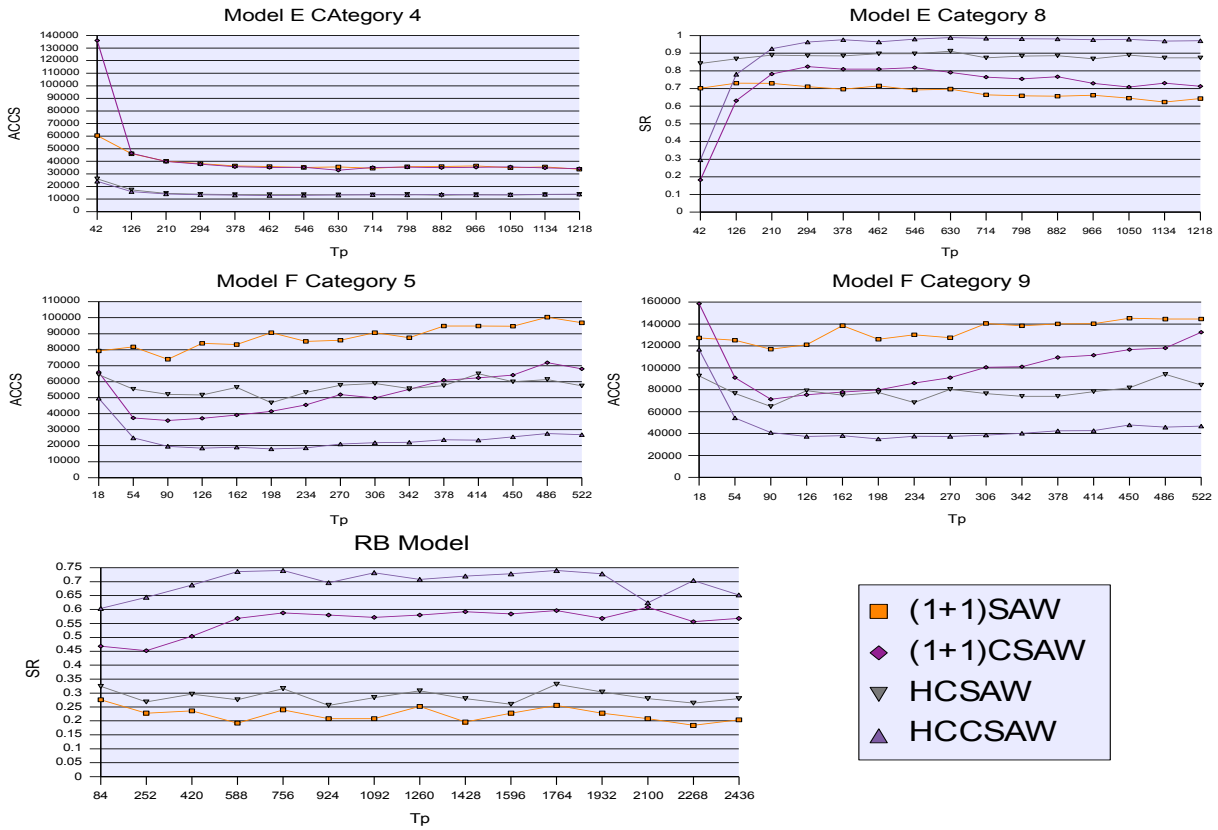


Fig. 1. The influence of T_p over all suggested weight adaptation algorithms on a selection of five different test-set categories and performance measures. The values of T_p are problem size dependent and range from $0.2n(m-1)$ to $5.8n(m-1)$. $maxCC$ is set to 1,000,000 conflict checks except for the RB model test for which $maxCC$ is 30,000,000.

rSAWEA was presented by the name SAWEA r2 as part of a comprehensive study of EAs for CSPs [11].

rSAWEA uses permutations of variables as individuals, and keeps a population of size 10. It mutates individuals by swapping two uniformly selected variables, and does not use a crossover operator. It uses a biased ranking parent selection [19] with a bias of 1.5, and a replace worst survivor selection operator. The SAW parameters it uses are $\Delta w = 1$, and it performs weight adaptation once every 25 generations, which is the traditional $T_p = 250$ (since population size is 10). As usual, weight adaptation is done using the best individual of the relevant population.

In order to evaluate individuals, a decoder assigns values to the variables according to their order in the permutation. Each variable receives the first value consistent with previous assignments, and otherwise considered violated. The individual is evaluated by summing the weights of unassigned variables. Even though using a weight per variable means using few weights this was not considered a problem since [5], [7], [8] showed no advantage for a constraint based SAW over variable based SAW.

The original version of the decoder that was used in a $(1 + 1)$ algorithm for the graph 3-coloring problem [5], always assigns values in the order of a static domain. This means that the first variable is always assigned the first value. It works well for the graph 3-coloring problem because

the symmetry of the values ensures the decoder does not exclude a solution. Moreover, for this problem there are only 3^n possible assignments which makes $n!$ permutations an overrepresentation, apparently without hurting performance. In CSPs, however, the values are usually not symmetric, and the domain size is m^n . Since m may depend on n , the $n!$ permutations may be an underrepresentation. In rSAWEA the decoder was enhanced by checking and testing several options for a dynamic assignment of the values [11]. The best choice turned out to be randomizing the order of domain values, and once every weight adaptation rotating domain values of unassigned variables so the first becomes last and all others advance one step forward. Assigning values with a dynamic domain ensures that all solutions are producible, yet this means that the actual assignment of the same permutation depend on runtime circumstances. In fact, rotating domain values means that we are also adapting the order of the values the decoder is using, and not only weights.

Better EAs, namely STLEA (Simple Tabu List Evolutionary Algorithm) and CTLEA (Conflict Tabu List Evolutionary Algorithm) were recently presented in [12] and [13] respectively by Craenen and Paechter. These algorithms use tabu lists to keep the algorithm from traversing already studied search-paths. Interestingly, the second and better version of the two algorithms, which is also the best known EA for CSPs, remembers conflicts rather than search space points.

TABLE I
MODEL F TEST-SET RESULTS

	HCSAW			HCCSAW			rSAWEA		CTLEA		STLEA	
	SR	ACCS	SDev	SR	ACCS	SDev	SR	ACCS	SR	ACCS	SR	ACCS
F-1	0.74	376	454	1	722	888	1	9665	1	1313	1	2576
F-2	0.80	8317	52410	1	3160	3541	0.98	350789	1	4670	1	67443
F-3	0.87	31421	97707	1	7861	7852	0.95	763903	1	20283	1	313431
F-4	0.90	52897	119578	1	15847	16949	0.97	652045	1	50745	1	397636
F-5	0.91	53486	124235	1	18337	20142	1	557026	1	94931	1	319212
F-6	0.91	70938	146524	1	24980	26842	1	715122	1	167627	1	469876
F-7	0.88	92896	152112	1	49671	57916	1	864249	1	239106	1	692888
F-8	0.86	102332	164354	1	47927	50222	1	1012082	1	254902	1	774929
F-9	0.96	70457	121603	1	37387	40773	1	408016	1	240046	1	442323

This may indicate the value of learning conflicts in order to push the search away from local optima. The details of both algorithms are rather complicated, and they will not be surveyed here.

Table I shows the results of running HCSAW, HCCSAW, rSAWEA, STLEA, and CTLEA on the model F test-set. The results for rSAWEA were originally published in [11]. The results for rSAWEA, STLEA and CTLEA are taken from [12], [13]. The time allowed for HCSAW and HCCSAW to complete their runs was 1 million conflict checks. The standard deviation of the conflict checks count of successful runs is also provided for HCSAW and HCCSAW, to allow an estimation of robustness. rSAWEA was allowed 100,000 evaluations of search points, and STLEA and CTLEA were allowed all the time needed to find solutions in all the runs, sometimes exceeding 1 million conflict checks.

The results for HCSAW show that even though it performs well on most runs, there are runs for which it fails even when the number of allowed conflict checks is over ten times the average needed for the successful runs. This is consistent with previous knowledge about the performance of EAs with population size 1 using variable or conflict based SAW [8]. Luckily, the problem does not occur when HCCSAW is used, and performance in terms of ACCS is further improved. Standard deviation figures of the conflict check count for HCSAW and HCCSAW show that conflict based SAW provides both a relative and an absolute improvement of robustness. A comparison of HCCSAW with rSAWEA shows that HCCSAW is over ten times, and up to a hundred times faster than rSAWEA. This is especially interesting when we remember that HC uses a population of size 1, and a straightforward search points representation, that were found not to work best with the usual weight adaptation schemes. The performance of HCCSAW is several times, and up to six times better than CTLEA, and even better w.r.t. STLEA.

HCCSAW is also able to find solutions in all the runs before STLEA and CTLEA. Tests showed that when each instance was run only 10 times per instance (similarly to the tests done for STLEA and CTLEA), 450,000 conflict checks were enough for HCCSAW to be successful on all runs. As reported in [12], [13], both algorithms needed more time to

always succeed in some of the categories; in most of them in the case of STLEA. This shows that weight adaptation, when used with weights for conflicts rather than for variables or constraints, is a very powerful technique.

V. CSPS OF VARYING HARDNESS

We turn to test the implications of using CSAW for soluble CSPS of varying hardness by using the model E test-set.

TABLE II
MODEL E TEST-SET RESULTS FOR $maxCC = 1,000,000$

	(1+1)SAW		(1+1)CSAW		HCSAW		HCCSAW	
	SR	ACCS	SR	ACCS	SR	ACCS	SR	ACCS
E-1	1	10801	1	10702	1	4073	1	4002
E-2	1	15382	1	15605	1	5785	1	5923
E-3	1	23192	1	24142	1	8816	1	8840
E-4	1	37195	1	37819	1	14157	1	13354
E-5	1	66797	1	61639	1	24898	1	23784
E-6	1	127283	1	120672	1	48065	1	45457
E-7	0.96	241927	0.98	239314	0.99	115867	1	97525
E-8	0.72	352572	0.80	370575	0.89	239675	0.96	246007
E-9	0.33	419616	0.44	433448	0.52	345121	0.58	374262
E-10	0.15	479161	0.21	486220	0.26	360991	0.28	441589

Table II shows the results for running the SAW and CSAW versions of the HC and (1 + 1) algorithms. $maxCC$ was set to 1 million conflict checks. Recall that problems become increasingly hard as we move from the first category to the last. The first five categories show little difference between SAW and CSAW on both algorithms. However, as we get closer to the phase transition, CSAW begins to outperform the constraint based SAW in terms of SR, which is the primary measure.

TABLE III
MODEL E TEST-SET RESULTS FOR $maxCC = 10,000,000$

	(1+1)SAW		(1+1)CSAW		HCSAW		HCCSAW	
	SR	ACCS	SR	ACCS	SR	ACCS	SR	ACCS
E-6	1	127283	1	120672	1	48065	1	45457
E-7	1	297102	1	249181	1	118642	1	100289
E-8	0.98	937903	1	605367	0.99	465234	1	292644
E-9	0.72	2.09e+06	0.94	1.90e+06	0.72	2.09e+06	0.98	1.29e+06
E-10	0.39	2.64e+06	0.75	3.20e+06	0.49	1.93e+06	0.90	2.74e+06

When the algorithms receive more time the effect of CSAW becomes more apparent. Table III shows results for the last five categories of the model E test-set with $maxCC = 10,000,000$. Apparently, there is a more dramatic difference in performance of the CSAW and SAW algorithms in the last category which is at the phase transition. Other categories show that the extra time helped CSAW widen the performance gap, yet in a more modest way.

VI. CONCLUSIONS

In this paper we explored weight adaptation with weights for every conflict rather than for every constraint or variable. We demonstrated that this significantly improves the performance of EAs solving hard CSPs, while having virtually no effect on easy CSPs. Moreover, a very simple EA employing conflict based SAW called HCCSAW turns out to be better than the best SAWing EA for CSPs, and even better than the best known EA for CSPs in general. This means that weight adaptation, when applied with a weight per conflict, is more powerful than previously implemented and perceived. HCCSAW proves superior even though it does not use the permutation representation, considered most suitable for SAWing algorithms solving CSPs and graph 3-coloring. It is also easy to program, and requires only robust parameters.

Conflict based SAW uses many more weights than constraint based SAW. Accordingly, more space is needed for them. The additional overhead in space and time needed for the most efficient runtime implementation of the weights is fairly insignificant even for relatively large CSPs. It consists of holding and initializing a matrix with an entry for all possible assignments of two variables. Otherwise there are no additional runtime costs. Therefore it is safe to say that using a conflict based SAW algorithm instead of a parallel SAW algorithm is never noticeably harmful, and with hard CSPs very beneficial.

The results obtained here are in contrast to the conception that adding more information, by adding more weights, does not improve the performance of SAWing CSP solvers. Although this is true when using a weight per constraint instead of a weight per variable, keeping a weight for each conflict does improve SAW. Consequentially, some of the results known to hold for variable and constraint based SAW algorithms solving CSPs should be rechecked using conflict based SAW. Among these are that EAs with a population larger than 1 are preferable, that refinement and decay are not effective, and that the order-based representation with a decoder is better than an integer representation of variable values.

Since a weight per clause was used for 3-SAT, which is parallel to using a weight per conflict, using our results for 3-SAT requires additional thought. For graph 3-coloring, the parallel of conflict based SAW is yet to be implemented and compared to previous suggestions.

Further research may include finding better conflict based SAW algorithms for CSPs. The contribution of randomness is unclear so a non EA approach might prove useful. In order to learn how effective conflict based weight adaptation is

for solving CSPs, local search algorithms that use it should be compared with efficient non local search algorithms for CSPs. Some form of a weight adaptation heuristic might also prove useful for non local search algorithms for CSPs.

REFERENCES

- [1] B. Selman and H. Kautz, "Domain-independent extensions to GSAT: solving large structured satisfiability problems," in *Proceedings of the International Joint Conference on Artificial Intelligence(IJCAI-93)*, Chambry, France, 1993, pp. 290–295.
- [2] B. Selman, H. J. Levesque, and D. G. Mitchell, "A new method for solving hard satisfiability problems," in *AAAI*, 1992, pp. 440–446.
- [3] J. Frank, "Weighting for godot: Learning heuristics for GSAT," in *AAAI/IAAI, Vol. 1*. MIT Press, 1996, pp. 338–343.
- [4] A. E. Eiben and Z. Ruttkay, "Self-adaptivity for constraint satisfaction: Learning penalty functions," in *International Conference on Evolutionary Computation*. IEEE Press, 1996, pp. 258–261.
- [5] A. E. Eiben, J. K. van der Hauw, and J. I. van Hemert, "Graph coloring with adaptive evolutionary algorithms," *J. Heuristics*, vol. 4, no. 1, pp. 25–46, 1998.
- [6] A. E. Eiben and J. K. van der Hauw, "Solving 3-SAT by GAS adapting constraint weights," in *IEEECEP: Proceedings of The IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, 1997, pp. 81–86.
- [7] B. G. W. Craenen and A. E. Eiben, "Stepwise adaption of weights with refinement and decay on constraint satisfaction problems," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds. San Francisco, California, USA: Morgan Kaufmann, 7-11 2001, pp. 291–298.
- [8] —, "An experimental comparison of SAWing EAs for a new class of random binary csp," in *Proceedings of 2002 Congress on Evolutionary Computation*. IEEE Computer Society, 2002, pp. 878–883.
- [9] F. Rossi, C. J. Petrie, and V. Dhar, "On the equivalence of constraint satisfaction problems," in *ECAI*, 1990, pp. 550–556.
- [10] J. Frank, "Learning short-term weights for GSAT," in *IJCAI (1)*, 1997, pp. 384–391.
- [11] B. G. W. Craenen, "Solving constraint satisfaction problems with evolutionary algorithms," Ph.D. dissertation, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, 2005.
- [12] B. G. W. Craenen and B. Paechter, "A tabu search evolutionary algorithm for solving constraint satisfaction problems," in *PPSN*, ser. Lecture Notes in Computer Science, T. P. Runarsson, H.-G. Beyer, E. K. Burke, J. J. M. Guervós, L. D. Whitley, and X. Yao, Eds., vol. 4193. Springer, 2006, pp. 152–161.
- [13] —, "A conflict tabu search evolutionary algorithm for solving constraint satisfaction problems," in *EvoCOP*, ser. Lecture Notes in Computer Science, J. I. van Hemert and C. Cotta, Eds., vol. 4972. Springer, 2008, pp. 13–24.
- [14] J. Gottlieb and N. Voss, "Adaptive fitness functions for the satisfiability problem," in *PPSN VI: Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*. London, UK: Springer-Verlag, 2000, pp. 621–630.
- [15] P. Prosser, "Hybrid algorithms for the constraint satisfaction problem," *Computational Intelligence*, vol. 9, pp. 268–299, 1993.
- [16] E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh, "Random constraint satisfaction: Theory meets practice," in *CP*, ser. Lecture Notes in Computer Science, M. J. Maher and J.-F. Puget, Eds., vol. 1520. Springer, 1998, pp. 325–339.
- [17] K. Xu and W. Li, "Exact phase transitions in random constraint satisfaction problems," *Journal of Artificial Intelligence Research*, vol. 12, pp. 93–103, 2000.
- [18] D. Achlioptas, L. M. Kirousis, E. Kranakis, D. Krizanc, M. S. O. Molloy, and Y. C. Stamatiou, "Random constraint satisfaction: A more accurate picture," in *Principles and Practice of Constraint Programming*. Springer, 1997, pp. 107–120.
- [19] D. Whitley, "The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best," in *Proceedings of the Third International Conference on Genetic Algorithms*, D. J. Schaffer, Ed. San Francisco, CA: Morgan Kaufmann, 1989, pp. 116–121.